UIUCDCS-R-78-935

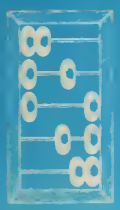
UILU-ENG 78 1728

AN IMPLEMENTATION OF A SYSTEM FOR THE FORMAL DEFINITION OF PROGRAMMING LANGUAGES

by

Brian Alfred Hansche

August 1978



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

AN IMPLEMENTATION OF A SYSTEM FOR THE FORMAL DEFINITION OF PROGRAMMING LANGUAGES

BY

BRIAN ALFRED HANSCHE

B.S., University of New Mexico, 1971
M.S., University of Illinois, 1976

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign 1978

Urbana, Illinois

AN IMPLEMENTATION OF A SYSTEM FOR THE FORMAL DEFINITION OF PROGRAMMING LANGUAGES

Brian Alfred Hansche, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1978

This paper describes a method for generating a table-driven interpreter for a programming language from a formal specification of its syntax and semantics. Such interpreters would be useful in verifying the correctness of formal specifications, and in providing experience with initial versions of experimental languages. The paper discusses existing formal specification methods and selects one method, based on a string replacement mechanism, as the basis for implementing a table-driven interpreter. A class of machines called Parse Tree Automata is defined. These machines are such that each state can be represented as a parse tree of a concrete program. An interpreter is then defined by a computation sequence of the Parse Tree Automaton. A method of constructing a table-driven interpreter based on these abstract machines is given and algorithms for reducing the number of transitions needed by the interpreter are supplied. The paper also includes a method of verifying that the formal specification is complete, well formed, and not redundant.

ACKNOWLEDGEMENT

The author gratefully acknowledges the aid and encouragement of Dr. G. R. Kampen in the development of this thesis. The author would like to express his appreciation of the Department of Computer Science at the University of Illinois for its continued moral and financial support. Special thanks are due to Dr. C. L. Liu for his help in the final preparation of this thesis. And finally, I would like to thank my wife, Margaret, for her patience and support during the development and preparation of this thesis.

TABLE of CONTENTS

LIST of FIGURES

CHAPTER 1

INTRODUCTION

The design of programing languages is a field in need of mechanical aids. Although some areas of language design, such as parser construction are supported by mechanical aids, there is no system which will support the entire design process. What is needed is a language design system which will take a formal definition of the language, verify the description, and then automaticaly implement the language. Such a system would free the designer of implementation details and let him concentrate on the design of the language. Once the formal definition of the language is complete and correct, the designer can then concentrate on an efficient implementation without concern over what should be implemented.

The thesis describes such a language design system. The system takes a formal specification of a language and generates a working interpreter for the language. This interpreter can then be used to study design decisions by running sample programs in the language. The system also aids in verifing the correctness of the formal definition of the language and checking the completeness of the definition.

This system takes the view that the basic definition of a language is its formal specification and not an implementation of the language. Once a formal specification of a language has been designed, the language can then be implemented at different installations and even on different computers and still be the same language. Programs written in this language can then be easily transported. Additionally, the formal description can be studied to answer questions about the language. Language design decisions may be studied by modifying the formal specification and implementing a test language using the system. In this way, language design decisions can be examined before the effort is put into building a compiler for a language.

The system can be used to design languages of any complexity. However, one of the goals of this system has been the easy design of small special purpose languages. Such languages can then be designed for the specific problem. The languages can be specially designed to use terms which

are natural to the problem and are in use by the proposed users of the language. Such special purpose languages are needed to support fields outside of computer science. Instead of forcing users to learn a programming language, we can design languages which are natural for the users. The proposed language design system is a tool which can be used to design such languages with a minimum amount of effort. Since the system includes an interpreter, once the formal specification is correctly specified, we will have a working language.

Several language design aids are already in use today. There are several different ways of automatically generating a parser for a context-free language. Indeed, this system uses an existing LALR(k) shift-reduce parse table generater, and a modified table driven parser. The major work which is necessary for a language design system lies in the automatic generation of language translators (compilers or interpreters). In existence today are systems which provide skeletons for the 'body of the compiler. The language designer must then fill in the details in order to have a working language translator. This system uses a table driven interpreter based on a new class of abstract automata, the parse tree automata. This automata is a modification of a string automata. Additionally, the entire area of formal specification of languages is still in need of much

clarification before a standard method of language specification will be accepted.

This thesis consists of two parts. The first part, chapters two through five, discusses the theory which underlies the language design system. Chapter two surveys the different techniques used to formally specify a programming language. Chapter three discusses string automata, while chapter four introduces a new type of automata, the parse tree automata. Chapter five discusses a method of partitioning grammars and then shows how such a partitioning can be used to verify the formal specification and to optimize the table driven interpreter. Finally, chapter six discusses the implemented language design system.

CHAPTER 2

FORMAL DEFINITION OF PROGRAMMING LANGUAGES

2.1 Formal Definitions

In order to understand a programming language, we must first give a definition for the language. Language definitions can range from a written description of what the language should do to ultramathematical definition. Regardless of the method of definition, the following problem must be addressed. Given an alphabet of symbols, S, the set S* is the set of all possible symbol strings that can be constructed from S. A language provides a subset, P, of legal programs. Moreover the language defines the meaning of each element of P. To define a programming language, we must

give some method of selecting the valid set of programs, P, and some way of assigning a meaning to each program in P. The definition of the syntax, or the form of the programming language, is the description of how to select the subset P. This description must describe both the context-free syntax and the context-sensitive syntax. Additionally, the formal definition must describe the semantics, or meaning, of each possible program in the language.

2.2 History

The formal description of the context-free portions of programming languages has been well understood for a number of years. Context-free grammars can be expressed in a number of forms. Early works on natural languages have given us good formalisms for specifying context-free grammars [Chomsky 1959] [Greibach 1965]. Programming languages have used formal methods of specifying their context-free syntax since the description of COBOL60 using a two dimensional approach to define the constructs of the language [Department of Defense 1960]. The first version of ALGOL used a metalinguistic notation introduced by Bacus to describe its context-free syntax [Backus 1959]. This normal form, BNF, is in wide use today.

Several different extensions of BNF include closure operators, optional clauses and even allow regular expressions. Perhaps the wide acceptance of BNF is due to the fact that it is clear and easy to use. Most modern definitions of programming languages include a description of the context-free syntax, usually in BNF or one of its derivatives. These formal descriptions can be used to define the context-free syntax of any language. Several systems use this type of formal description to generate the information necesary to parse the language.

The techniques for the formal definition of the context-sensitive syntax and the semantics of programming languages are less developed. Early definitions of the semantics of programming languages were usually given in prose or even entirely omitted. Often the only definition of the complexities of the language were "defined" by a particular implementation of its compiler. The Vienna Definition Language (VDL) was used to describe the syntax and semantics of PL1 in 1968 [Lucas, Lauer, and Stigleituer 1968]. ALGOL68 was defined using W-grammars in 1968 [van Wijngaarden, et al 1968]. Since then several differnt techniques for the formal specification of semantics have been developed. These techniques range from explicit methods which generate all valid programs to ultra-abstract techniques relying on recursive function theory. The reason

for several different approaches is that the definition of semantics is not as straight foreward as the definition of context-free syntax. Each of the different methods has its strong points and its weaknesses.

2.3 Goals of Formal Definitions

Regardless of the method of specification, several goals are desirable. These inculde:

Completeness. There should be a complete description of the language. The formal specification should be able to answer all questions about the syntax, the semantics, and implemation restrictions.

Clarity. The method of description should be easy to understand. The description must be balanced between too much and too little abstraction. An excessive amount of abstraction can hide the details of the language behind the abstraction mechanism.

The lack of sufficient abstractions can hide the meaning behind the sheer bulk of the specification. Whatever formal description method is used should be easy to learn and natural to use.

Realism. The description method must include some mechanisms for expressing the restrictions which are imposed by the real world. Such implementation restrictions as finite storage space and word sizes are important details which must be expressed. The abstract description must be able to express these details.

Taken together these goals aim the formal description towards a complete understandable description of a programming language. Such a description would include context-free syntax, context-sensitive syntax, and semantics of the language. The description method should be able to describe implementation restrictions.

Also desirable in the formal description is the separation of context-free syntax, context-sensitive syntax, and semantics. This separation allows the form and meaning of a language to be separated. Indeed most descriptions separate the context-free syntax from the rest of the description. The context-sensitive requirements are often described with the semantics.

2.4 Techniques of Formal Definition

2.4.1 Context-free Syntax

Several different types of specification of the context-free syntax are available today. These include the two-dimensional representation used to define Cobol, BNF, the flow diagrams used with Pascal, and others. They are all equivalent and capable of describing any context-free language. Probably the most common method is the Backus Normal Form [Backus 1959]. In BNF, nonterminals of the grammar are enclosed in brackets (<,>), terminals are written as themselves, and a production is indicated by "::=". Several production rules with the same left hand side may be

grouped together using the alternation operator ("|"). For example the context-free syntax of simple expressions may be written:

```
<Exp>     ::= <Factor>    | <Exp> * <Factor>
<Factor>  ::= <Term>      | <Factor> + <Term>
<Term>    ::= ( <Exp> )   | <Number> | <Id>
```

where <Number> is the nonterminal which derives all valid constants, and <Id> derives identifiers.

2.4.2 Context-sensitive Syntax

Unlike context-free syntax, the formal methods of describing the context-sensitive syntax are less developed. Several different approaches have been used. One approach is to specify a grammar which only generates those programs which conform to the context-sensitive requirements. Another technique is to define a translation process which translates a valid context-free program into an intermediate form. During this translation, the context-sensitive requirements may be checked.

An example of the first technique is the description of ALGOL68 [van Wijngaarden, et al 1968]. ALGOL68 was defined by a W-grammar. A W-grammer specifies two sets of rules which can be combined to form a possibly infinite set of production rules. These production rules generate only those programs

which meet the context-sensitive requirements.

As an example, consider the definition of a simple declaration list. Each identifier name can be any single letter of the alphabet. There is an additional context-sensitive requirement that no two identifiers may be the same letter. In a W-grammar for the declaration list, the first set of rules, called the metaproductions, might be:

```
TAGS      ::  TAGS , TAG;
              TAG.
TAG       ::  ALPHA
ALPHA     ::  a; b;  . . .  z.
ALPHABET  ::  abcdefghijklmnopqrstuvwxyz.
EMPTY     ::  .
ALPHSETY  ::  ALPHAS;
              EMPTY.
ALPHAS    ::  ALPHA;
              ALPHAS ALPHA.
```

In the metaproductions, the symbol "::" is used to separate the left and right sides of the metaproductions, the symbol ";" is used as an alternation operator, and the symbol "." is used to terminate a metaproduction. In this example, the metanotion TAGS generates a list of one-letter identifiers separated by the comma symbol. The metanotion TAG can generate any letter (an underlined symbol is used to represent a terminal).

The context-sensitive requirements are introduced by coupling the metanotions with a second set of rules, the hyperrules:

```
dcl              : TAGS dcl sequence.
TAG dcl sequence : TAG.
TAGS , TAG dcl sequence :
                   TAGS dcl sequence , TAG
                   where TAG is not in TAGS.
where TAG is not in TAG2 TAGS :
                   where TAG is not TAG2
                   where TAG is not in TAGS.
where tag is not in TAG2 :
                   where TAG is not TAG2.
where TAG is not TAG2 :
                   where TAG precedes TAG2 in ALPHABET;
                   where TAG2 precedes TAG in ALPHABET;
where TAG precedes TAG2 in ALPHSETY TAG ALPHSETY
TAG2 ALPHSETY3 : EMPTY.
```

In the hyperrules, the symbol ':' is used to separate the right and left sides of the rule while ';' and '.' are used to indicate an alternative and the end of the rule. In any hyperrule, we may replace all occurences of a metanotion by any of its productions. The resulting set of rules, which may be infinite, can then be used to produce all valid strings of terminals. In addition to producing a valid terminal string, these rules may result in a dead en which cannot be reduced. These dead ends correspond to programs which violate the syntax. Consider the two possible dcl-lists, a,b and a,a. In the first case the derivation sequence is:

dcl
TAGS dcl sequence
TAGS , TAG dcl sequence
TAGS dcl sequence , TAG where TAG is not in TAGS
TAGS dcl sequence ,b where b is not in TAGS
a dcl sequence ,b where b is not in a
a dcl sequence ,b EMPTY
a dcl sequence ,b
a,b

Thus we see that the valid dcl-list can be derived. Actually, the metarules and the hyperrules combine to form a set of production rules for dcl:

dcl :: a | b | ... z | a,b | a,c ...

This set of rules includes a righthand side for the list a,b. However, there is no possible rule which will derive a,a from dcl. When we try to derive an invalid dcl-list (a,a), we run into a dead end:

dcl
TAGS dcl sequence
a,a dcl sequence
a dcl sequence ,a where a is not in a

Here we cannot go any further since the clause "where a is not in a" cannot be reduced. Thus we see that W-grammars generate only those programs which conform to the context-sensitive syntax.

The more common technique of specifying the context-sensitive requirements is to specify a translation pahse to validate the context-sensitive requirements. The Vienna Definition Language defines the context-sensitive requirements in this fashion. The translator component of the abstract VDL-machine is actually the definition of the

context-sensitive requirements. To define a dcl-list of unique identifiers, we first define an arbitrary dcl train:

```
dcl  :  alpha | alpha , dcl
```

and then the translation function:

```
valid-dcl-list(dcl) =
          there does not exists x1,x2 such that
          (x1≠x2) and
          is-c-id(x1(dcl)) and
          is-c-id(x2(dcl)) and
          x1(dcl)=x2(dcl) )
```

Here, x1 and x2 are functions, called selector functions, which select an arbitrary son of the node dcl. Therefore they select any id in the dcl-list. The function is-c-id returns true only if its argument is a valid identifier name. The function valid-dcl-list returns true only if there do not exist two different selectors which select equal identifiers, i.e., if there are no duplicate names in the dcl-list. This is a context-sensitive check to see if there are two identifiers that are identical.

2.4.3 Semantics

Perhaps the most important part of any language is the semantic meaning. It is therefore unfortunate that formal methods for specifying the semantics of a program have been so long in developing. Even today, the most frequent method of describing the semantics of a programming language is a written description in a natural language such as English. Existing formal methods exhibit a wide range of formalism and abstractness, ranging from specification by compiler [Garwick, 1966] to specification by mathematical model [Tennent, 1976]. These methods can be loosely grouped into three categories: devolutional functional, and interpretive.

2.4.3.1 Devolution

Devolutional methods provide a translation algorithm which can map any program in the language being defined, into another, equivalent program in a known language. The known language, called the target language, may be a high level language, a machine language, or even a subset of the language being defined. When the target language is machine code, the formal definition of the language is its compiler. When defining the language in terms of itself, the language

is extensional [Irons, 1970]. For example, we can define an exchange operator (:=:) in terms of the normal assignment operator (:=) by mapping the exchange operator into a subset of the language:

```
a :=: b     ::     (LOCAL T; T:=A; A:=B; B:=T)
```

The disadvantage of devolution is that the target language must be defined in some way. This is not too much of a problem if the target language has already been formally defined. If the target language has no formal definition, some errors may arise from different interpretations of the target language.

2.4.3.2 Functional

Functional and axiomatic methods tend to be implicit rather than constructive. A functional definition of a language is specified by defining mappings of the syntactic constructs of the object language into their abstract "meaning" in a mathematical model. Typically, the meaning of any program (prog) in the object language is defined by a mapping, M:

```
M: prog -> (I->O)
```

where I is the possible set of inputs to the program, and O is the possible set of outputs.. The axiomatic method

([Hoare, 1974]) is based on proving assertions about the programming language. These proofs are based on predicate calculus and involve some steps which must be proved by the user. For example, to define the meaning of two statements executed sequentially, we might use:

```
semstm(stm1 ; stmt2 : rho) <=> A1 [*] B3
  <= Provable<A1:B1> and
     semstm(stm1 : rho) = B1 [stm1:rho] A2 and
     Provable<A2:B2> and
     semstm(stm2 : rho) = B2 [stm2:rho] A3 and
     Provable<A3:B3>
```

Here, the Ai's and the Bj's are assertions about the program. The function semstm are logical predicates about the action of individual statements. Provable<A:B> is a logical predicate which is true if and only if the user can prove that A derives B. The symbol * is simply a shorthand method of respecifying a string which is used in the predicate and in the proof of the predicate. In this case, * = "stm1 ; stmt2 : rho". The notation Ai [*] Bj means that if Ai holds before a statement *, and we execute *, then Bj must hold. In this example, we are proving that if A1 holds before executing stm1;stm2 then B3 must hold after executing the statements.

2.4.3.3 Interpretive

Interpretive methods define a language by exhibiting an interpreter that transforms the current state of a computation into its successor state. The current state includes a representation of the program being executed and a memory component. The program may be represented by a character string corresponding to the concrete program ([Kampen 1973]) or by an abstract object representing the parse tree, as in the Vienna Definition Language. The interpreter is then defined as a transition function on these states. The semantic meaning of a concrete program is then defined by applying the interpreter to the program. The resulting sequence of states (its computation sequence) and especially the final state in the computation is taken to be the semantic meaning of the program.

2.5 Uses of Formal Descriptions

The primary purpose of any formal definition is to provide information about a language. This information can be used for several different purposes. It can be used to answer questions about the language which arise from several different sources. Users of the language need to know what is permitted in the language and what implementation restrictions are imposed on them. Compiler writers need to know what should be implemented and what restrictions they need to impose on the language.

A formal description of languages is useful in business for writing contracts which specify exactly what specifications are needed in a language. Without detailed information about the language, it is difficult if not impossible to write transportable programs.

The formal definitions are useful in proving the correctness of programs. In fact, it is possible to prove general theorms about the language. For example, Kampen showed that it is impossible to have dangling references in SIBYL and that under certain restrictions on the use of loops, every program will eventually terminate.

While the present technology is not quite up to automatic generation, a formal notation will be necessary for the automatic validation of programs, and for the automatic generation of compilers. Formal notations are also useful in studying the general theory of programming languages.

2.6 Drawbacks of Formal Descriptions

In spite of recent work on formal descriptions, most techniques suffer to some degree from several shortcomings. Among the problems are:

Hard to learn. The metalinguistic termonology and techniques of formal definitions must be powerful enough to define any language. Consequently they are all complex, difficult to learn, and hard to use.

Difficult to write clear concise descriptions. Due to the complexity of programming languages, it is hard to write descriptions which do not omit any details.

Hard to modify. Many modifications propagate their changes through the entire description. This makes even the most trivial changes a difficult task.

Unsupported by mechanical aids. Due to the size and complexity of the descriptions, mechanical aids for maintaining and editing are desirable. Even more desirable is a mechanical system to verify the formal description.

Even though several languages have been formally defined ([Lucas, Lauer, and Stigleituer 1968], [van Wijngaarden, et al 1968]), the use of formal definitions have met with mixed reactions. There is considerable user resistance to the use of formal definitions, probably due to the shortcomings listed above. This resistance will only be overcome by developing the definitions to make them easy to use. Mechanical aids for editing and verification should be introduced.

CHAPTER 3

STRING AUTOMATA

3.1 Introduction to String Automata

In this chapter we will introduce the string automata. The parse tree automata (discussed in chapter 4) are an extension to the string automaton. In fact both use the same metalanguages L1 and L2. In this chapter we will discuss these metalanguages and describe the application of transition rules written using them.

An abstract machine called a string automaton was introduced by Kampen [Kampen 1973] as a means of presenting a formal language definition in a clear and concise manner. String automata permit a modular approach where related parts of the description are placed in small easily understood modules. These modules can then be linked together in a network to define a complete language. The string automata approach uses a string matching and replacement algorithm to define an interpreter for the language being defined. The syntax of the program is expressed using a metanotation that resembles BNF. The interpreter is used to specify the context-free requirements and assign the semantic meaning to the program. The interpreter is defined using a set of one or more transition rules. These rules specify the transition function of the string automaton. Since the string automata have the power of Turing machines, they can in principle be used to define the semantics of any programming language.

3.2 Definitions and Notation

Formally, a string automaton is a 4-tuple, SA = (V,N,S,T) where V is a finite set of symbols, N is a positive integer and S is a set of N-tuples of strings from V*. T is a mapping, T:A -> B, where A and B are subsets of the set S. The members of S are called states. When T is a function, the string automaton is deterministic otherwise, it is nondeterministic.

If s and t are states and if t = T(s) then t is called the successor of s and the relationship is indicated by s -> t. A sequence of states, s(Ø),s(1),...,s(i) such that s(i) -> s(i+1) is called a computation and s(Ø) is called the initial state. We write s ->* t if and only if there exists a computation s(Ø),s(1),s(2),...,s(i) where s(Ø) = s and s(i) = t. If every state s(i) in a computation has a successor state s(i+1) = T(s(i)), then the computation is said to be infinite or nonterminating, otherwise, the computation halts in some terminal state, s(k), which is in the set of halt states, S - A.

Let R = (r<1> ,r<2> ,...,r<n>) be an n-tuple of objects called registers. Define an instance I of a string automaton M = (V,N,S,T) as the ordered pair (M,R). When I is in state s = (s<1> ,..., s<n>), the string s<i> is called the contents of register r<i>, and r<i> is said to contain s<i>.

The terms state and computation also apply to instances of string automata.

Note that a string automaton of n registers may be easily converted to a string automaton of one register simply by adding a new symbol to the alphabet and using this new symbol to separate substrings of the new machine. For example, if s = (s<1>,s<2>) then a single register machine can be constructed by introducing a new symbol, $, and defining the new state to be s' = s<1>$s<2>. The new transition function T' is then defined on V + {$} instead of on V x V.

3.3 Metalanguages

The specification of a string automaton is written in two metalanguages, L1 and L2. L1, the syntactic metalanguage, describes a set of N grammars, one for each register of the string automaton. These grammars define the set of valid states, S. The metalanguage L2 is used to describe a set of semantic descriptions which define the transition function, T, of the string automaton.

3.3.1 Syntactic Metalanguage

The metalanguage L1 is used to define a set of N grammars which describe the possible contents of each of the registers. Each grammar consists of a set of rules of the form

name => expr

where name is the name of a syntactic class (a non-terminal symbol of the grammar) and expr is an expression involving syntactic class names and strings of characters (terminal symbols) over the alphabet V. By convention, terminal strings will be underlined while syntactic class names will be capitalized. The empty string will be represented by e. The operators of the metalanguage are "|", "+", and "*" which are taken to mean, "or", "one or more occurences", and "zero or more occurences", respectively. The operator "|" has the lowest precedence, while "*" has the highest. Parentheses may be used to group operands and to overide operator precedence. Note that blanks are not significant and that grammars in metalanguage L1 may be indented to increase readability.

An optional list of variable names may be associated with any syntactic class. These variable names will represent any single element of the corresponding syntactic class. Variable names are akin to typed variables in a programming language; the type in this case is just the syntactic class, which specifies what values are permitted for the variable. By convention, variable names are written in lower case with an optional integer suffix. Often, the variable name will be the same as the name of the syntactic class of which it is a member. To associate a list lst of variables with a syntactic class n defined by an expression exp, we will write

lst: n=> exp

For example, a class Exp of integer expressions is defined in figure 1. The syntactic class Exp is a class of simple integer expressions with the operators "+" and "*". The string variable exp denotes any instance of this class. For example, exp might be 2*3. Note that the right hand side of the rule for the non-terminal Part includes an alternative which is empty. The symbol e indicates that the empty string is a member of the syntactic class Part.

```
exp: Exp          => Operand Part
part:  Part       => e | Operator Operand Part
op:    Operator   => + | *
x,y:   Operand    => Nil | Digit+
         Nil      => Ø
         Digit    => Ø | 1 | 2 | 3 | ...  | 9
```

Syntactic Description of Simple Expressions

Figure 1

Each of the rules in the metalanguage L1 has a single nonterminal on the left hand side of the rule and a right hand side which is a sequence of terminal strings and nonterminal symbols. Therefore, L1 describes the class of context free languages (Type 1). L1 does include rules whose righthand side is empty (erasing rules), but this is still equivalent to the class of context free languages. The context sensitive requirements of the language being defined will be described along with the semantics.

3.3.2 Semantic Metalanguage

The semantic description specifies the context-sensitive constraints of the programming language being defined, as well as describing an interpretive algorithm for assigning a semantic meaning to any program. A semantic description for a language is a string automaton that executes programs in the language.

A semantic description provides an algorithm for checking the context-sensitive requirements of the language by executing programs. The context-sensiive checking can be done by having the interpreter print an error message and halt whenever a context-sensitive requirement is not met. A semantic meaning is assigned to a program by executing the

string automaton with an initial state which corresponds to the program. The final result of this execution (the halt state of the string automaton) is the semantic meaning of the program.

The semantic description consists of a set of one or more transition rules that define an interpreter for the language. Informally, a state is compared with all of the transition rules. If the current state matches a rule, then the rule is evaluated and a new state is formed. Each tranition rule has the form

ruleid: (p<1>,p<2>,...,p<N>) -> (e<1>,e<2>,...,e<N>)

Here the p<i> are patterns, each of which is a sequence of terminal strings and string variables. The e<i> are string expressions and are composed of terminal strings, string variables, and string-valued functions of string expressions. The patterns are used to specify which states the rule will be applied to, while the expressions indicate how to construct the next state. String variables which appear in some expression must also appear in some pattern in the same transition rule. The pattern p<i> is a template for the contents of the register r<i> of the string automaton. These templates are used in the matching process with the terminal strings representing constant portions and the string variables representing those portions of the register, r<i>, which may vary within the limits of the corresponding

syntactic class. The expression e<i> prescribes the contents of r<i> in the successor state.

For example, a possible semantic description for the evaluation of simple expressions in the class Exp is:

E1: (x '+' y part) -> (Plus(x,y) part)

E2: (x '*' y part) -> (Times(x,y) part)

Where x,y, and part are string variables defined by the metalanguage L1 (see Figure 1). Plus and Times are functions which return integers represented as strings. For example, if we have a current state of 2+3*2 then the computation defined by rules e1 and E2 is the sequence

2+3*2

5*2

1Ø

3.4 Evaluating the Transition Function

Let us consider a deterministic string automaton, M, and a current state, S. The successor state, S', is determined in the following manner:

(1) Determine the first transition rule, Tj, whose pattern matches the current state S.

(2) Evaluate the expression of Tj. The resulting string is the successor state, S'.

To construct the successor state, we need to know which transition rule matches the current state and how to construct a new state from this rule.

3.4.1 The Matching Process

Given a transition rule,

Tk = (p<1>, p<2>,...,p<N>) -> (e<1>, e<2>,...,e<N>)

and a current state s = (s<1>,s<2>,...,s<N>), the matching process is as follows.

Starting with i=1;

(1) Match the string s<i>, which is the contents of register r<i>, against the pattern p<i>. Matching is done by parsing the string s<i> using a topdown parse with backup. If the

parse succeeds, then s<i> matches p<i> and we associate each matched substring of s<i> with the corresponding string variable in p<j>.

(2) If the parse succeeds and i<N, set i=i+1 and process the next register of the pattern. If any of the string variables of p<i> have been assigned a value by a previous match, replace those variables with the corresponding values. Go to (1).

(3) If the parse succeeds and j=N, then T matches s.

(4) If the parse fails, then T does not match state s.

3.4.2 Evaluation of Expressions

When a current state, s, matches a transition rule Tj, we may construct a new state by evaluating the expression of Tj. To compute the value of each register, r<i>, we first replace every string variable of e<i> with the value which was bound to that variable during step (1) of the successful match. Any functions in e<i> are then evaluated. The final result of the expression is the concatination of all the strings in e<i>. These strings are the results of function calls, constant strings (terminals) or the values of string variables.

3.5 Deterministic String Automata

A string automaton can be either deterministic or nondeterministic depending on the transition mapping, T. If T is a function, then the string automata is deterministic, otherwise, it is nondeterministic. Both the deterministic and nondeterministic automaton have equivalent computational power since they both are capable of simulating a Turing machine. Since a deterministic machine is easier to understand, we will restrict the transition mapping, T, so that it is a function. The transition mapping is specified

in tabular form using the metalanguage L2. There are two restrictions on the construction of new states which insure that T is a function. These restrictions are:

(1) During matching, the contents of a particular register are matched against a pattern using a top down parse. If the programming language is ambiguous, it is possible to match the same string in several different ways which may result in several different successor states. If there are several different parses of the same string, restrict the transition function to use only the match which uses the longest string for the first string variable. If there are two possible matches with the longest string possible matching the first string variable, choose the one which uses the longest string for the second string variable. Continue in this fashion, choosing the match which uses the longest string first, until only one possible match remains.

(2) It is possible for several different tranition rules to match the current state. Restrict the string automaton by using only the first (topmost) transition rule which matches. Then the successor state is formed by evaluating the expression of this transition rule.

Together these two restrictions insure that T is a function. The first chooses only one possible way to match a registers and the second causes only one transition rule to be applied. Of course there are several different ways of restriction the string automaton so that it is deterministic. Each different restriction may produce a slightly different deterministic string automaton.

3.6 Networks of String Automata

Small modules defined using a string automaton can be linked together in several ways. This allows a large definition to be broken up into small easy to understand parts. For instance, one may separate out the control structures, expression evaluation, and input/output of a language into separate modules and link them together. Although we may link modules together in many different ways, the following operations are useful. Let T1 and T2 be string automaton, define:

```
T = T1 o T2     iff  T(s)=T1(T2(s)) for all s,
T = T1 & T2     iff  T(s)=T1(s) when T1(s) is defined
                          =T2(s) otherwise,
T = T1*         iff  T(s1)=s2 where s1->*s2 and
                               s2 is a halt state in T1,
T = T1*n        iff  T=T1 when n=1
                     T=T1 o T1*(n-1) otherwise.
```

The composition operator, o, corresponds to function composition. To evaluate T1oT2(s), we first apply T2 to the state s and get an intermediate state s'. Then we apply T1 to s' to get the result state of T1oT2(s). The operator & corresponds to appending the rules of module T2 to the rules of T1. If state s matches a transition rule in T1, then T1(s) is defined and we will apply T1. If s doesn't match a transition rule in T1, we will reach the appended rules for a transition rule form T2 and a transition rule form T2 will be applied.

The operatators * and *n correspond to repeated application of a module, either until a halt state is reached (*) or for exactly n applications (*n). Repeated application is used to generate the final result of a computation sequence. If we have a string automaton, SA, which defines a programming language, and we have a program, prog, in that language, then the final result of executing that program is SA*(prog).

The preceeding operations have assumed that both T1 and T2 are defined on the same set of registers, R. Even when the modules are defined on disjoint sets of registers, we may define operations which combine the modules. Let R be a set of registers and let P and Q be subsets of R. Define:

T(R:Q)(s) = s' iff T(q)=q' where

s'<i> = q'<j> and s<i> =q<j> when R<i> =Q<j>

and s'<i> = s<i> otherwise.

This definition simply extends a string automaton, T, defined on a set of registers, Q, to an automaton defined on a larger set, R. The contents of the reqisters belonging to Q are transformed according to T, while the registers not in Q are left alone. We can also combine two automaton defined on different sets of registers, P and Q:

T(R) = T1(P) + T2(Q)

iff T=(T1(R:P)oT2(R:Q)) & T2(R:q) & T1(R:p)

This defines T to be a string automaton whose successor state is defines as follows. Apply T2 to the contents of Q and then apply T1 to the state induced by the new contents of P.

These operations allow us to define small easy to understand modules and then connect them together in a network. For example, Kampen defines a high level programming language, SIBYL, in this manner [Kampen 1973]. First define the modules:

E Expression evaluation

R	Primitive values, booleans,numbers,strings
D	Data structures- records, arrays
V	Memory management- stores, fetches
P	Procedures
C	Control structures
S	Self extension

In fact, the module for memory management, V, is actually composed of two modules, one to find variables in the memory and one to replace values in the memory or in expressions. The module V is the composition of these two modules (V=V'oFind).

These modules can then be linked together to define a processor for SIBYL:

```
Proc = E & R & D & (V'oFind) & P & C & S.
```

Kampen also defined a complete installation as a network of concurrently executing modules. For example, a configuration with an operator console, a tape unit, a printer, and three processors all sharing the same memory can be defined:

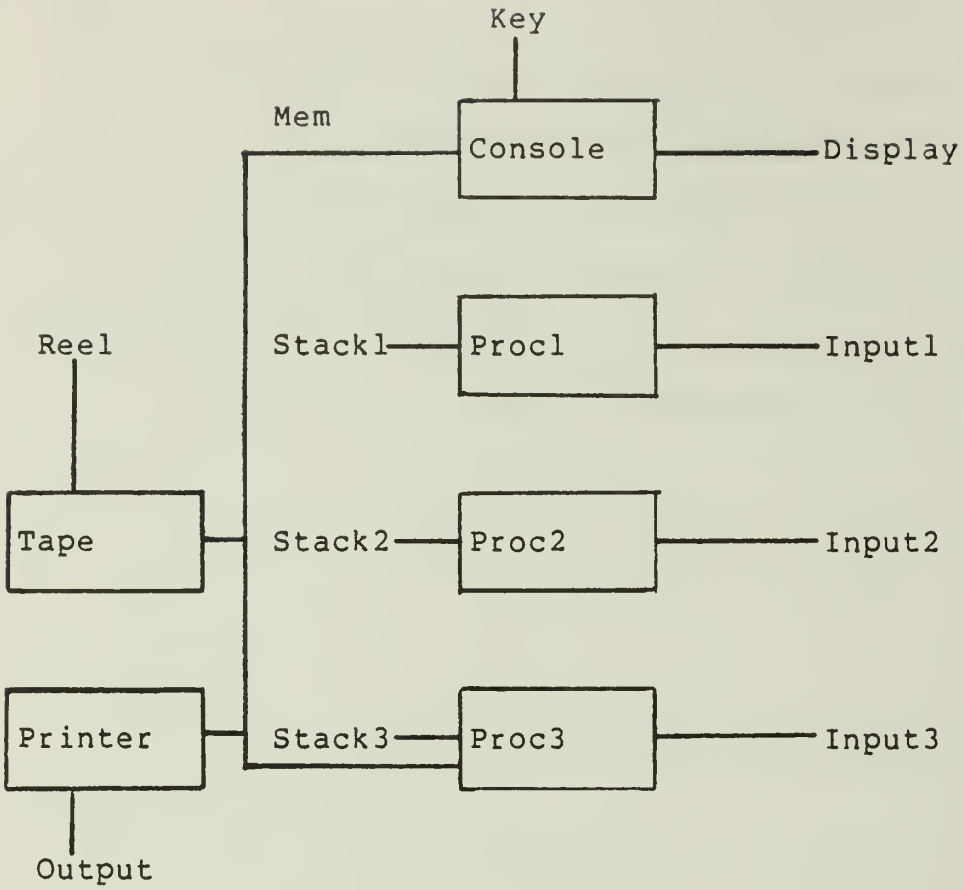
```
Inst(r) = I1 + I2 + I3 + C + T + P where
     I1 = Proc(Mem,Stack1,Input1),
     I2 = Proc(Mem,Stack2,Input2),
     I3 = Proc(Mem,Stack3,Input3),
     C = Console(Mem,Display,Input1,Keys),
     T = Tape(Mem,Reel),
     P = Printer(Mem,Output),
```

R =

(Mem,Display,Keys,Reel,Output,Stack1,Stack2,

Stack3,Input1,Input2,Input3).

Since all of the modules share the register Mem, they all share the same memory. However, each of the processors has its own input and stack. The configuration described by Inst is diagrammed in figure 2.



Multiprocessor Configuration

Figure 2

CHAPTER 4

PARSE TREE AUTOMATA

4.1 Discussion of String Automata

The string automaton is a natural choice for the formal description of programming languages. It indicates a way of implementing an interpreter for the language. All one has to do is provide a parser for the language and build an interpreter which uses the transition function described using L2. However, this type of implementation would be slow for several reasons:

(1) During every match, the current state which represents a program in the language, must be parsed. This repetitive parsing is unnecessary

since we have a parse tree of the program after each application of the transition rule.

(2) To construct the next state we need to match the current state against all the transition rules. However, many of these matches may be unnecessary. We can use information about the current state to eliminate many of the matches from consideration.

To overcome these two problems, we shall modify the string automaton to work on parse trees instead of strings. We will need to parse the the program only to calculate the first state of the computation sequence. We will also use the parser to initially construct parse trees for the patterns and expressions of the transition rules. We may use some information about the structure of the parse trees to speed up the matching process. If we are trying to match val op val2 against the parse tree for 56425+67742 we need only look at the top nodes of the tree to determine if the match succeeds or fails. In a string automaton, we would have to build and examine the parse tree of the entire string. Moreover, we will be able to use information about the structure of each transition to eliminate the unnecessary matching.

4.2 Definitions and Notation

An extended context-free grammar G = (SS,TS,P,Start) is a 4-tuple where NS is the set of non-terminal symbols, TS is the set of terminal symbols, and Start is a nonempty subset on NS called the starting symbols. The set of multiple starting symbols has been introduced to allow several different grammars to be merged into one extended grammar. The vocabulary, V, is the union of the nonterminal symbols, NS, and the terminal symbols, TS. The intersection of NS and TS must be empty. P is a mapping from NS to V* If A => A1 A2 ... An, is a production rule of P, and if x and y are strings of V* then x A y => x A1 A2 ... An y. This indicates that the string x A1 A2 ... An y can be derived from x A y by an application of a production rule. A derivation is done by replacing any nonterminal by the right hand side of any production rule for that nonterminal. A series of derivations, x => y => ... => z, may be written x =>* z.

The language Generated by G, denoted L(G), is defined to be:

L(G) = { x | A =>* x and x is in TS* and A is in Start }.

L(G) is the set of all terminal strings which can be derived from any element of the set or starting symbols by a series of applications of the transition rules.

A sentential form is a string, sf, from V* such that S =>* sf for some element S of Start. In general, a sentential form may be used to derive other sentential forms and to eventually produce terminal strings (sf =>* x, with x in L(G)). Therefore (if there are no useless rules in the grammar) a sentential form is simply an intermediate step in the derivation of a terminal string.

A parse tree for a string y, in L(G), is a labeled tree which satisfies the following requirements:

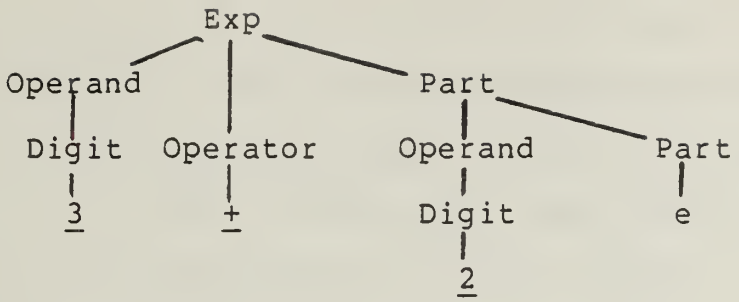
(1) The root of the tree is labeled with a starting symbol.

(2) The internal nodes are labeled with nonterminal symbols.

(3) The leaves of the tree are labeled with terminal symbols. The concatenation of all the leaves of the tree forms the string y.

(4) If a node labeled A has sons labeled A1, A2, ..., An, then A => A1 A2 ... An must be a production rule in P.

If p is the parse tree for a string y, then we say that y is the result of p. For example,



is the parse tree for the string 3+2 (see figure 1 for a deffinition of the grammar). Recall that e denotes the empty string. The set of parse trees of the terminal strings in L(G) is denoted P(G).

P(G) = { p | x is in L(G) and x is the result of p }

A section of a parse tree is a sequence of nodes, {N1,N2,...,Nk} ,which may be internal or external nodes of the parse tree, such that:

(1) No node, Ni, is the ancestor of any node Nj.

(2) For every leaf, Li, in the parse tree there is a node Nk such that either Nk is the leaf Li, or Nk is an ancestor of Li.

A section is simply an intermediate result in deriving the terminal string from the start symbol (S =>* N1 N2 ... Nk =>* L1 L2 ... Lm). Note that every section is also a sentential form. For example,

Operand + 2 Part

Digit Operator Operand Part

Operand + 2

3 + 2

are all sections of the parse tree of 2+3.

A parse tree automaton is a 4-tuple, (N,G,S,T), where N is a positive integer, G is an extended context free grammar defined on a finite set of symbols, and S is a set of N-tuples of parse trees of strings taken from L(G). T is a mapping, T:A -> B, where A and B are subsets of the set S of states. When T is a function, the parse tree automaton is deterministic, otherwise, it is nondeterministic. The mapping T has domain and range S which is contained in P(G1)xP(G2)x...xP(Gn).

If s and t are states and if t = T(s) then t is the successor of s and we write s => t. A sequence of successors, s(Ø) => s(1) => ... => s(n) is called a computation sequence with s(Ø) as the initial state.

If R = (r<1>,r<2>,...,r<n>) is an N-tuple of registers, then an instance, I, of a parse tree automaton, M = (N,G,S,T), is the ordered pair (M,R). When I is in state s = (s<1>,s<2>,...,s<n>) the parse tree s<i> is called the contents of r<i>, and r<i> is said to hold s<i>. The contents of a register s<i> is described by all strings derivable form one of the starting symbols of the extended context-free grammar. A computation sequence of instances is a sequence I(Ø),I(1),...I(N) such that the contents of the registers of I(j+1) are the successors of the contents of the registers of I(j).

We may convert an multi-register automaton to a single register automaton by defining a new grammar which 'links' the extended context-free grammar together by introducing a new start symbol and a rule which produces all the original start symbols from the new start symbol. Define the production rule R' to be S' => S1$S2$...$Sn where $ is a new nonterminal symbol intrduced to prevent ambiguity, and S1,S2,...,Sn are the starting symbols in the extended context-free grammar, G. A new grammar can then be created as G' = (V+{{S'},$},P1+P2+...+Pn+{R'},{S'}) where V is the set of all symbols in G and Pi is the set of production rules from the grammar Gi. The new automaton of one register is then M' = (1,G',S',T').

4.3 Specification of Parse Tree Automata

A parse tree automaton can be specified in a similar manner as a string automaton. The description of the automaton consists of two parts, the syntactic description and the semantic rules. We use the metalanguages L1 and L2 to describe the syntactic form of the automaton, and to define the transition function (the semantic rules). These parts are similar to the declarations and body of a programming language. The syntactic description (type

declarations) define the valid states of the automaton while the semantic rules (program) define a series of actions on the valid states.

The metalanguage L1 is used to present the syntactic description. This description is a set of production rules for the grammars. Each grammar describes the permissible contents of one of the registers of the parse tree automaton. An example of a syntactic description is shown in figure 1.

The semantic rules define the transition function of the parse tree automaton. The semantic rules are defined by a series of transition rules using the metalanguage L2. In the semantic rules, each pattern element (p<i>) and each expression element (e<i>) must be restricted to a sentential form of the grammar Gi. In these sentential forms, nonterminal symbols will be represented by their variable symbols. For example, if v is a variable name for the syntactic class V then x v y would be used to represent the sentential form x V y. In the implementation of the parse tree automaton, it will be necessary to construct parse trees for these sentential forms. Therefore, we must modify the syntax to include these variables. If we have a grammar G1 which defines the context-free syntax, we will modify it by adding new production rules. For every pair, (V,v), of syntactic class names and variables we will add the production rule V => v to G1. If x and y are terminal

strings and if x V y is also a sentential form, then x v y is also a terminal string since x V y => x v y. In this mannar, we modify the grammars to allow us to construct parse trees from sentential forms.

As an example, let us use a parse tree automaton to define a simple pocket calculator. The automaton will have three registers, one which represents an internal stack, one for the display and one for the keyboard (input) of the calculator. The description of the context-free langugaes which describe the possible contents of these registers is given in figure 3. The transitions rules of the automaton are shown in figure 4.

Figure 3 describes three grammars, one for each register of the parse tree automaton. The transition function is then described in figure 4. The transition function Calc has domain and range Stack x Display x Input. A transition from p to q is defined by the transition rules in the semantic rules. If the parse tree p matches the pattern of a rule, Ti, then the expression of that rule is evaluated to yield a new parse tree, q.

```
stk:      Stack          => e | Operand Operator
dis:      Display        => e | Operand
y:        Input          => e | Key Input
key:        Key          => Operator | Digit
op:         Operator     => + | *
val:        Operand      => Digit | Operand Digit
dig:          Digit      => 0 | 1 | ... | 9
```

Syntactic Description of a

Pocket Calculator

Figure 3

```
calc1: (stk , e , dig y )   -> (stk , dig , y )
calc2: (stk , val , dig y ) -> (stk , val dig , y)
calc3: ( e , val , op y )   -> (val op , e , y)
calc4: (val + , val2 , y )  -> ( e , Plus(val,val2) , y)
calc5: (val * , val2 , y )  -> ( e , Times(val,val2) , y)
```

Semantic Rules of a

Pocket Calculator

Figure 4

4.4 Construction of the Successor State

Given a state, s=(s<1>,s<2>,...,s<n>), of a parse tree automaton PT=(N,G,S,T), the successor state s is calculated as follows; starting with j=1:

(A) Consider the transition rule

Tj = (p<1>,p<2>,...,p<n>) -> (e<1>,e<2>,...,e<n>)

Set i=1;

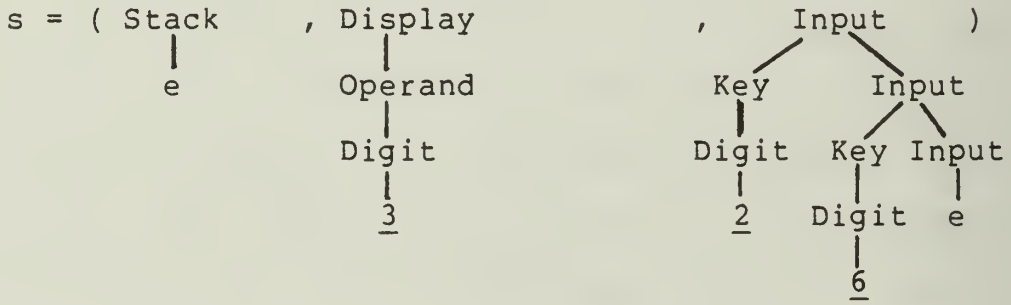
(1) Match p<i> against s<i>. A match occurs only if the sentential form p<i> is a section of the parse tree s<i>. During this matching, a variable may be either undefined or may be bound to a particular subtree of s<i>. If a variable is undefined, then any subtree of the appropriate syntactic class may match the variable. This value is then bound to the variable and subsequent occurences of the variable will be defined. If a variable has already been defined, then the only permissible match is an identical subtree.

(2) If a match succeeds, set i=i+1 and if i<N, go to (1). If the match fails, set j=j+1 and repeat the process with a new transition rule (go to (A)). If the transition rules are exhausted, then there is no possible successor state and the current state is a halt state.

If all the patterns, p<i>, match the parse trees, s<i>, for every i, then the current state matches the transition rule under consideration.

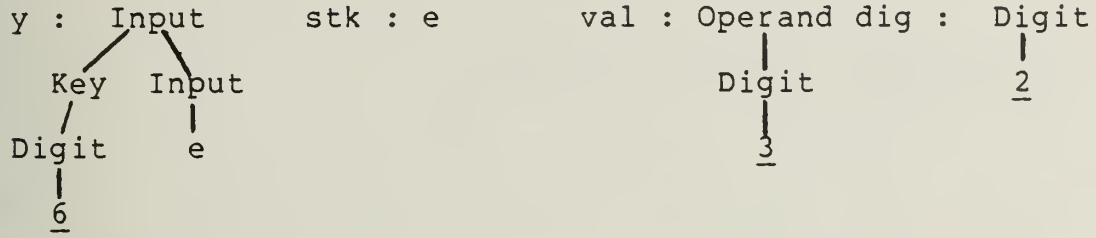
For every transition rule that matched the current state, construct a new state by evaluating the expression of the rule. To evaluate the expression, first replace all variables by the values which were bound to them during the successful match. Next evaluate all functions (each function must return a parse tree) and reconstruct a parse tree using the values of the variables, the results of the functions, and the parse trees of the expression.

For example, consider a possible state of the pocket calculator:

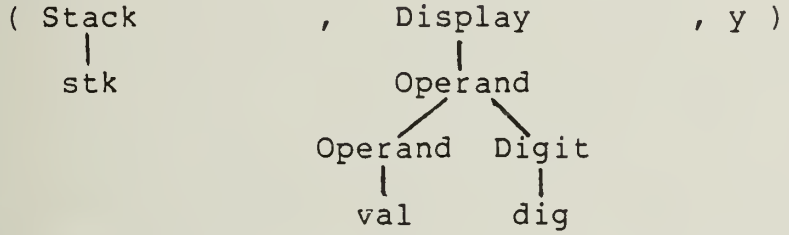


If we tried to match transition rules calc4 and calc5 to this state, the matching process would fail since the first register of the state is empty (see figure 4). Rule calc3 will match the first and second registers and will bind the parse tree of the string 3 to the variable val. However the rule will fail to match the third register since 'op y' is not a section of s<3>. Rule calc1 will match the first

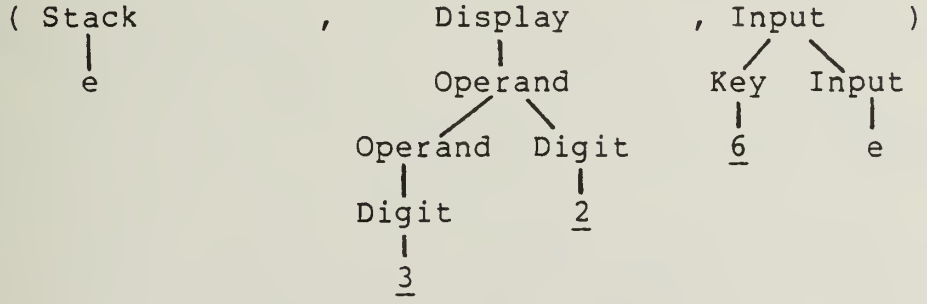
register (and bind the empty parse tree to the variable stk) but will fail to match the second register. The only rule which does match is calc2. After matching, the values of the variables of calc2 are:



A new parse tree is then formed by taking the parse trees of the expression:



and replacing the variables, stk, val, dig, and y by their values. The resulting new state is:



This is equivalent to a string automaton starting with the state s = (, 3 , 26) and matching against the rules in figure 4. The resulting state woule be (, 32 , 6). In the

parse tree automaton, we get the same result except that the contents of register r<i> is the parse tree of the string s<i>.

4.5 Comparison with String Automata

Since we may define both a parse tree automaton, PT, and a string automaton, SA, using the same description written in the metalanguages L1 and L2, we would expect that the automata would be closely related. We say that a parse tree automaton, PA, is equivalent to a string automaton, SA, if and only if for every possible state s of SA, there exists a state p of PA such that p is the parse tree of s and for all direct successors, t, of s there exists a q, such that q is a direct successor of p and q is the parse tree of t. If p is a state of PA, and if s is a state of SA such that p is the parse tree of s, we say that p and s are equivalent states.

Theorem: If a string automaton, SA=(V,N,L(G),T) and a parse tree automaton, PA = (N,G,P(G),T) are defined using identical syntactic descriptions and identical semantic rules and if all the grammars in G are nonambiguous, then PA is equivalent to SA.

Proof. Since the set of states of SA is L(G), for any state, s, of the string automaton, there exists a parse tree in P(G). Therefore, for every state s in SA, there is an equivalent state in PA. Let p be the equivalent state of t and let t be any successor of t (s -> t in SA). Since s is a successor of t, s must match some transition rule Ti=(p<i>->e<i>) such that t is the evaluation of the expression e<i>. The pattern p<i> must be a sentential form of G since the rule Ti is a rule of a parse tree automaton. Since p<i> matches s and since p<i> is a sentential form, p<i> must be a section of a parse tree of s. Since the grammar is unambiguous, there is only one parse tree of s and this is the tree p. Therefore, rule Ti of TA will match the state p. In evaluating the expression, e<i>, in SA, we use certain substrings of s as the values of the string variables. In evaluating e<i> in the PA, we will use the parse trees of the same values. Since the expression e<i> is a sentential form and since the expression derives the string t, e<i> must be a section of a parse tree in P(G). However, since the grammar is unambiguous, and since the expression e<i> is used to produce the successor of p, we must have a parse tree q, such that p -> q and q is the parse tree of t. Therefore, for any state, s, of SA there is a state, p, of PA such that p is the parse tree of s and that for any direct successor, t, of s, there exists a parse tree q in PA such

that q is the parse tree of t and p -> q. Hence, the parse tree automaton, PA, is equivalent to the string automaton, SA.

If one of the grammars in G is ambiguous, then the string automaton and the parse tree automaton defined using the same description may not be equivalent. Consider the automaton defined by:

x: X => e | X a | Z

y: Y => e | a Y

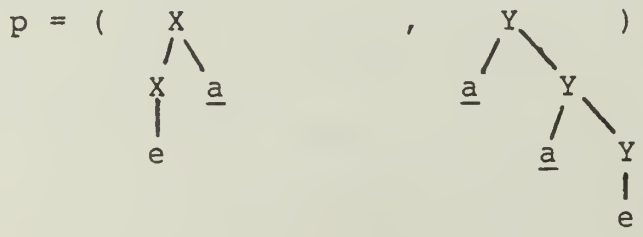
z: Z => aa

r1: (z , y) -> (a , y)

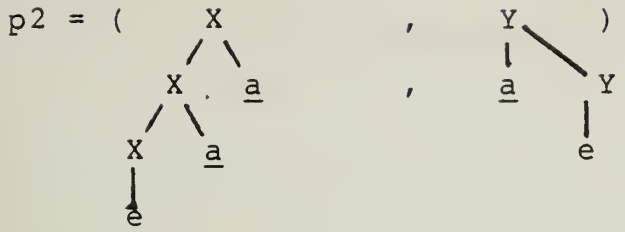
r2: (x , a y) -> (x a , y)

In a string automaton, the state s = (a , aa) has the successor, s1 = (aa , a), which in turn has the successor, s3 = (a , a). State s3 is the result of applying rule r1 to s2.

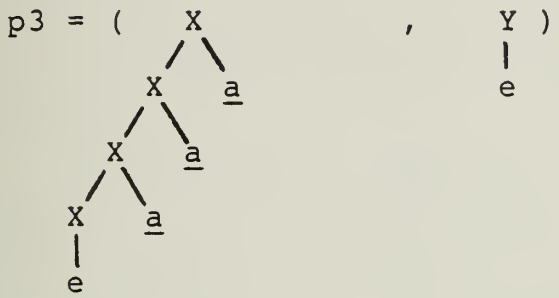
In the parse tree automaton, the equivalent state to s is:



Which has the successor state:



However, the successor of p2 results from the application of r2:



Rule r1 fails to match state p2 since the first register does not hold a parse tree whose syntactic class is Z.

Actually, there is a parse tree of (aa , a) which will match r2 (and not r1) but it is not the result of applying the transition function to p2. In a string automaton, the lexemes a a can merge together to form the lexeme aa, this is impossible in a parse tree automaton. Once a lexeme is recognised, it cannot merge with any other lexeme to form a different type of lexeme.

4.6 Formal Description of Languages using a Parse Tree Automata

We can use a parse tree automaton to formally define a programming language. The context-free syntax of the language can be defined using the syntactic description of the parse tree automaton. The context-sensitive requirements of the language, and the semantics of the language can then be defined using the semantic rules of the parse tree automaton. A program in the language will first be parsed using the context-free grammar defined in the synactic description. The context-sensitive requirements of the language can then be checked using a set of transition rules. Finally, the program may be 'executed' using the transition rules to as a definition of the semantics of the language.

As an example, consider a simple language which declares a variable and then assigns the constant 1 to the same variable. The syntactic description of the language is then:

```
        Pgm     => e | dcl Var ; Var := 1
        Mem     => e | Var : Val
    v:  Var     => a | b | c
        Val     => u | 1
```

The semantic rules are then:

```
r1:  ( e , dcl v , v := 1 )   -> ( v : 1 , e )
r2:  ( e , dcl v1 , v2 := 1 ) -> error
```

The first rule correspond to the execution of a valid program. If the variables in the assignment is the same as the variable in the declaration, then the value 1 is stored in the memory and the program is erased. During the matching, the variable v will be assigned a value. After this assignment, only the same value will match the second occurence of v in rule r1. If the variables do not match, then the second rule will be applied, and an error will be indicated.

It is desirable that the grammars in the syntactic description be unambiguous. If this is the case, then the parse tree automaton is equivalent to the string automaton defined using the same rules. We will also make the parse tree automaton deterministic by only applying the first (topmost) transition rule that matches.

We may use the same operations as a string automaton to link together several different small modules of transition rules. The operators:

T = T1 o T2	composition
T = T1 & T2	concatenation
T = T1*	closure
T = T1*n	n applications of T1

defined on string automaton are also defined with the same meaning on a parse tree automaton. In addition, the operations T(R:Q)(p) and T1(P)+T2(Q) are also defined in

exactly the same way. See chapter 3 for more details about these operators.

To use a parse tree automaton to define a large language, we first break up the definition into several smaller modules. For example, we might define the expression evaluation separately from the definition of memory. Therefore, the module that defined expressions does not need to know the details of how the memory is represented. The two modules are then linked. For example, the module Expr might define expression evaluation, while Fetch might describe how the value of an identifier is recovered. To evaluate (Expr*)o(Fetch*)(x+y) we would first replace the variables x and y by their integer values (Fetch*(x+y)) and then evaluate the expression (Expr(2+3)).

CHAPTER 5

PARTITIONING CONTEXT-FREE GRAMMARS

5.1 Intersection of Transition Rules

We say that two transition rules are independent if their domains are disjoint and are dependent if their domains intersect. This means that two independent rules will never match the same state. If we wish to rearrange the transition rules to improve their readability, we may do so by interchanging adjacent independent rules. If we only interchange adjacent independent rules, we will not change the meaning of the semantic moule. The domains of two transition rules are disjoint if and only if the sets of terminal strings derivable form the patterns of the rules are disjoint. For example, if we have the rules,

calc1: (stk , e , dig y) -> (stk , dig , y)

calc2: (stk , val , dig y) -> (stk , val dig , y)

calc3: (e , val , op y) -> (val op , e , y)

Then the first two rules are dependant since their domains both include the parse tree of

e , e , dig y

Therefore, the order between rules calc1 and calc2 must be maintained. On the otherhand, rule calc3 is independent of both calc1 and calc2. We are free to interchange calc2 and calc3 if we so desire.

In order to calculate the dependancy relations between rules of a parse tree automaton, we need to know if, for any two transition rules, there is any state which will match both rules. If we have two patterns, p1 and p2, defined using a grammar, G=(NS,NT,P,S), we may introduce two new symbols, S1 and S2, and the production rules R1: S1 => p1, and R2: S2 => p2. We can now define the strings derivable from p1 and p2 as the languages of two grammars. Define G1 = (NS+S1,TS,P+R1,S1) and G2 = (NS+S2,TS,P+R2,S2). Now {x|p1 =>* x} = {x|S1 =>* x} = L(G1); and {y|p2 =>* y} = {y|S2 =>* y} = L(G2).

We may extend the definition of L(G) to define the set of strings derivable from any sentential form. For any sentential form, sf, define L(sf) = {x|sf->*x}. Similarly, if Q is a set of sentential forms, define L(Q)={x|q->*x for q in Q}.

We may now rephrase the question of the intersection of two transition rules. There exists a state of the parse tree automaton which matches both the transition rules p1->e1 and p2->e2 if and only if the intersection of L(p1) and L(p2) is non-empty. Since both L(p1) and L(p2) are context-free languages, this is in general an unsolvable question. However, by restricting the grammar, G, we can determine if two sentential forms of G do intersect.

We will use the intersection algorithm to construct a partition of the strings in L(G). The partition will be constructed in such a manner that each pattern and each expression of the semantic rules will be the union of some of the blocks of the partition. We may then use the blocks to determine if rules may be rearranged. The blocks can be used to construct a finite state machine which models the parse tree automaton. This machine can then be used to improve the efficency of the interpreter by eliminating unnecessary matches. By treating the matching rules like a decision table, we can also use the blocks of the partition to test the semantic rules for completeness and for redundancy.

5.2 Intersection of Sentential Forms

If we require the grammar G to unambiguous, we can then determine if there is any string matched by two sentential forms, p1 and p2.

Lemma 1 If A and B are sentential form in an unambiguous grammar, and if L(A) int L(B) is nonempty, then there is a sentential form C such that A =>* C and B =>* C. Moreover, C will derive any string which is derivable form both A and B (if A =>* x and B => x then C =>* x).

Proof Assume A and B are nondisjoint sentential forms and x is a string which can be derived from both A and B. Then S =>* A =>* x and S =>* B =>* x. Since the grammar is unambiguous, there is only one possible parse tree of x. Therefore, both A and B are sections of the same parse tree. Thus A and B match the same string if and only if:

A<1> A<2>...A<i> => B<1> B<2>...B<j>

B<j+1> B<j+2>...B<k> => A<i+1> A<i+2>...A<1>

A<1+1> A<1+2>...A<m> => B<k1><k+2>...B<n>

.
.
.

B<s+1> B<s+2>...B<t> => A<r+1> A<r+2>... A<u>

where

A = A<1> A<2>...A<u>

B = B<1> B<2>...B<t>.

The sentential forms A and B both derive a common section, C, of the parse tree of x. This section is composed of nodes from both A and B (the nodes on the right side of the derivation above). These nodes are the nodes of A and B which are farthest from the root. Thus A =>* C, and B =>* C and any string which may be derived from both A and B may also be derived form C.

We can determine if two rules T1 and T2 are independent by examining their patterns, p1 and p2. If L(p1) int L(p2) is empty, then the rules are independent. We may test the intersection of L(p1) and L(p2) by using lemma 1. If we can construct a sentential form C such that p1 =>* C and p2 =>* C then the rules are dependent. If such a sentential form does not exist, then the rules are independent. Thus, the rules are dependent if and only if:

```
p1<1> p1<2>...p1<i>          => p2<1> p2<2>...p2<j>
p2<j+1> p2<j+2>...p2<k>      => p1<i+1> p1<i+2>...p1<1>
p1<1+1> p1<1+2>...p1<m>      => p2<k+1> p2<k+2>...p2<n>
                                .
                                .
                                .
p2<s+1> p2<s+2>...p2<t>      => p1<r+1> p1<r+2>...p1<u>
```

where

```
p1 = p1<1> p1<2>...p1<u>
p2 = p2<1> p2<2>...p2<t>.
```

If the rules are dependent, then the common sentential form C

will be:

C = p1<1> p1<2> ... p1<i>
p2<j+1> p2<j+2> ... p2<k>
p1<i+1> p1<i+2> ... p1<m> ...
p2<s+1> p2<s+2> ... p2<t>

5.3 Partitions

Now consider a set of sentential forms, Q={Q<i>|1<=i<=m} such that:

L(Q<i>) int L(Q<j>) is empty for i≠j

L(G) = {x|Q<i>->*x for some i such that 1<=i<=m}

Such a set of sentential forms is called a partition of the language L(G). Additionally for any sentential form sf we may define a partition Q of L(sf) as a non-intersecting set of sentential forms Q<i> such that L(sf) = {x|Q<i>->*x}. If we have a partition Q we can refine the partition by replacing an element Q<i> by a partition of that element. Let Q'={Q'<j>} be a set of sentential forms such that:

L(Q'<i>) int L(Q'<j>) is empty for i≠j

L(Q<i>) = union L(Q'<j>)

Then a refinement of Q is Q - Q<i> + union(Q'<j>). Note that a refinement of a partition is also a partition.

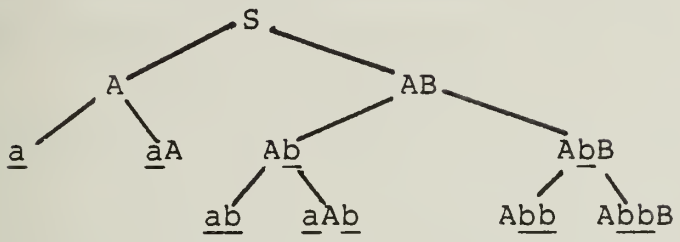
Consider the grammar G2:

S => A | AB

A => a | aA

B => b | bB

and consider the tree:



The root of the tree is a partition ({x|S->*x}=L(G2)). At each internal node, we have replaced a nonterminal by all of its possible right hand sides to obtain its sons, which is simply a refinement of the partition. Thus any section of the tree can be arrived at by a series of refinements and is therefore a partition of the grammar, G2. Note that there are several trees of this type. At a node we may refine the partition by replacing any nonterminal by its right hand side. This may yield many different trees. A section of any of these trees is also a partition. If fact this tree is actually a subtree of a (possibly infinite) tree whose leaves are precisely the sentences of the langugae L(G). The set of all partitions is precisely the set of all sections of this tree and all trees obtained in a similar fashion.

5.4 Constructing Partitions

Consider a sentential form, sf, of a grammar G, and a partition Q={Q<i>}. We may construct a new partition, Q' such that L(sf) = L(union Q<k> for some subset of elements of q'). Such a partition is called a refinement with respect to the sentential form sf. To refine a partition, Q, with respect to a partition, sf, we start with the set Q' empty.

While Q is nonempty:

(1) Let Q<i> be an arbitrary element of Q. Remove Q<i> from A.

(2) If Q<i> intersect sf is empty, add Q<i> to Q' and go to (1).

(3) If sf =>* Q<i>, add Q<i> to Q' and to to (1).

(4) Since sf int Q<i> is non-empty, but sf ≠>* Q<i>, we must refine Q<i>. By lemma 1, there is a sentential form, C, which is composed only of elements of Q<i> and sf such that Q<i> =>* C and sf =>* C. Let Q<i,j> be the first nonterminal of Q<i> which does not appear in C. Refine Q<i> by replacing this nonterminal by all possible right hand sides, and add each element of the refinement to Q. Go to step (1).

This algorithm continues to refine elements of the partition Q until either their intersection with sf is empty, or sf derives the element of the partition. Note that steps 2 and 3 each remove exactly one element from Q while step 4 adds an arbitrary number of elements to Q. When this algorithm halts, Q' is a partition with respect to the sentential form sf.

Theorem If the grammar, G, is nonambiguous, then the partition construction algorithm will halt with a refined partition, such that the sentential form, sf, is the union of some of the elements of the refined partition.

Proof. Elements are added to Q' in either step 2 or step3. If they are added in step 2, then their intersection with sf is empty. If they are added in step 3, then L(Q'<i>) is contained by L(sf) since sf derives Q'<i>. Since Q' is a partition of L(G), every string in L(G) must be in some set, Q<i>. Consider the set L(sf) and the set Y = {Q<i>| Q<i> =>* y for some y in L(sf)}. Clearly L(sf) is contained by L(Y). Moreover, each element of Y must have been added to Q' during step 3 of the algorithm. Therefore, sf => Q'<i> for all Q'<i> in Y and hence L(sf) contains L(Y). Since L(sf) both contains and is contained by L(Y), we must have L(sf) = L(Y), and therefore, L(sf) = L(union of Q'<i> such that Q'<i> is in Y). Hence, if the algorithm terminates, it will terminate with a refinement of Q with respect to sf.

We will now prove that the algorithm halts. Consider an application of step 4. We have some partition element Q<i> = q<i,1> q<i,2> ... q<i,n> and a sentential form, sf = sf<1> sf<2> ... sf<m>. Since the intersection of Q<i> and sf is nonempty there must be some string x which they both generate, and since the grammar is unambiguous, there is only one possible parse tree for x; sf and Q<i> are sections of this parse tree. Since sf $\not>$* Q<i>, there must be some nodes of Q<i> which are ancestors of nodes of sf. Step 4 replaces one of these ancestor nodes and introduces new partitions to the set Q. These new partitions, Q<j>, are either contained in sf, have an empty intersection with sf, of lie in the same parse tree as sf. In the first two cases, the new partitions will be removed from Q in step 2 or step 3 of the algorithm. In the latter case, we have the same parse tree, p, with some nodes of Q<j> lying above the sentential form, sf. This section must be entirely contained by the original partition Q<i> and must have some nodes which are ancestors of some of the nodes of sf. Since there are only a finite number of possible ancestor nodes of sf, and since every application of step 4 introduces new partitions which have nodes lower in the parse tree, after a finite number of applications of step 4 we will have replaced Q<i> with a refinement whose elements are either contained by sf, or have an empty intersection with sf. Therefore, the algorithm will terminate.

We may also refine a partition with respect to a set of sentential forms. Simply refine the partition with respect to the first sentential form and then refining the refinement with respect to the other sentential forms. If Q is a partition, define Ref(Q|sf) = the refinement of Q with respect to the sentential form sf. Also define Ref(Q|{sf1,sf2,...,sfn}) = Ref(Ref(Q|sf1)|{sf2,...,sfn}).

5.5 Example of Partitioning a Grammar

As an example, consider applying the partitioning algorithm with the sentential form bb, the starting set of partitions {S}, and the grammar G2:

S => A | AB

A => a | aA

B => b | bB

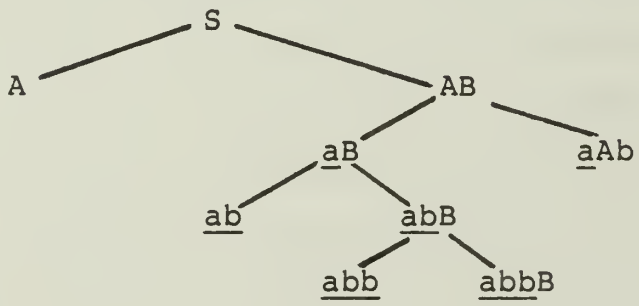
The partitioning algorithm the following sets:

Q	Q'	next Q<i>	step
{S}	empty	S	4
{A,Ab}	empty	A	2
{Ab}	{A}	Ab	4
{aB,aAB}	{A}	aB	4
{ab,abB,aAB}	{A}	ab	2
{abB,aAB}	{A,ab}	abB	4
{abb,abbB,aAb}	{A,ab}	abb	3
{abbB,aAB}	{A,ab,abb}		

and with two more applications of step 2:

Q = empty Q' = {A,ab,abb,abbB,aAB}

A tree for this derivation is:



Each internal node of this tree corresponds to an application of step 4 of the partitioning algorithm, which replaces a nonterminal by the right hand side of its production rules. The sons of a node are the strings which may be derived by replacing a nonterminal by its right hand sides. The leaves of the tree correspond to elements added to Q' during either

step 2 or step 3. The leaves are the elements of Ref({S}|abb). We may take another sentential form, say aa, and further refine the partition to obtain the set

Ref({S}|{abb,aa}) = {a,aa,aaA,ab,abb,abbB}

by refining each leaf of the tree with respect to aa.

This process may be continued until the partition has been refined with respect to all desired sentential forms. Notice that if Q is a refinement with respect to sfl, then Q' = Ref(Q|sf2) is also a refinement with respect to sfl. In refining Q, the partition elements are never joined together. Therefore if Q was a refinement with respect to sfl, then after replacing some elements of Q with their refinements, Q' is still a partition, and L(sf) = L(union Q'<i> for some subset of Q').

5.6 Uses of Partitions

Let us consider a set T of transition rules for a string automaton. The transition rules give us a set P={p<i>} of patterns and a set E={e<i>} of expressions. Let G=(NS,TS,P,S) be the grammar which defines the parse trees of the patterns and the expressions. Let U = Ref(S|P+E) and define a function Index over the elements sf of P+E, such that,

Index(sf)={i|sf =>* u<i> for u<i> in U}.

Then for every element of P (or E) there is a set of integers, Index(p<i>), which index the elements of the partition which compose p<i>. Thus L(p<i>)=union L(u<j>)for j in index(p<i>).

Consider the application of the transition rules of a parse tree automaton to a state s. We must match s against all the partitions of T. Since s is the result of evaluating some expression e<i>, we do not need to match s against all the patterns of T. We need only to match s against those patterns of T which have an non-empty intersection with e<i>. Let Next(e<i>)={j|Index(p<j>) int Index(e<i>) is nonempty}. Thus we need only test the rules Tj where j is in Next(e<i>).

Even though the parse tree automaton operates on a possibly infinite set of states, we can use the information in the partitions to define the underlying finite state machine (UFSM) of a parse tree automaton. A state of the underlying machine corresponds to the set of trees which will be matched by a particular rule of the parse tree automaton. Thus, the states of the UFSM correspond to collections of elements of the partition. A state of the underlying machine corresponds to a set of trees of the parse tree automaton.

State<i> ={t|t is a parse tree of some stiring
in L(G) and t matches p<i> and for j<i,
t does not match p<j>}

There is a transition from State<i> to State<j> if and only if there is some tree, t, in State<i> such that the successor tree in the parse tree automaton, s, is in State<j>. Thus the successor states of State<i> reflect the set Next(e<i>). The states which are successor states of State<i> are the states State<j>, where j is in Next(e<i>). The halt states of the underlying machine are those states which contain a tree that does not match any pattern of the parse tree automaton. The intitial state, State<Ø>, will have transitions to all the states which correspond to rules which can be initially applied in the parse tree automaton. If no information is supplied about the initial state of the parse tree automaton, then the initial state of the

underlying machine will have transitions to all the other states.

In general, we have restricted a parse tree automaton so that the only rule applied will be the first rule matched. This restriction keeps the parse tree automaton deterministic. Suppose we have two transition rules Ti and Ti+1 and that we which to interchange the order of these two rules. (Such an interchange might make the rules more readable.) We can make the interchange if and only if the domains of the rules do not intersect. Thus we may interchange rules Ti and Ti+1 if and only if Index(p<i>) int Index(p<i+1>) is empty. If the intersection of the index sets of two rules is empty, then the rules are independent and may be interchanged.

We may use the index sets to identify rules which operate on the same type of states. We may wish to group these rules together in a separate module. Such dependent rules may be identified by examining the index sets of all the rules.

Perhaps the most important use of partitions is in verifying the rules. The partitions can be used to find redundant rules and to check the completeness of a module. A set of transition rules is similar to a decision table. The patterns correspond to the truth values in the decision table while the expressions correspond to the actions. We can

check the transition rules for redundancy and completeness in a similar mannar to the way a decision table is checked for completeness and redundandy.

A rule, Ti, is redundant if and only if for every state, s, which matches p<i>, there is another rule Tj such that j<i and s also matches p<j>. If a rule is redundant, then in a deterministic automaton it will never be applied. In general, a redundant rule indicates that some error has been made in the specification of the rules. We may identify redundant rules using the partitions. A rule will be redundant if and only if for every integer k in Index(p<i>), there is a rule, Tj, such that j<i and k is in Index(p<j>). To test an entire module for redundant rules, we simply start at the top of the module with the set Used=empty.

For i=1 to number of rules do

(1) The rule Ti is redundant if there is no element i of Index(p<i>) such that i is not in Used.

(2) Used = Used + Index(p<i>)

Once we have identified a redundant rule, we may remove it form the module without effecting the transition function.

A set of transition rules defined on L(G) is complete if and only if for every string s in L(G) there is some transition rule Ti such that p<i> matches s. If a set of rules is complete, then there is no state which doesn't correspond to a transition rule. Note that a set of transition rules is complete if and only if for every i there is a transition rule with pattern p such that i is in Index(p). We may test for completeness usint the same algorithm which detected redundant rules. A module is complete if after executing the redundancy check, the set Used=Index(S).

CHAPTER 6

LANGUAGE DESIGN SYSTEM

6.1 The Implemented System

A language design system based on parse tree automata has been developed. This system verifies the correctness of a formal specification and generates an interpreter based on this specification. The system can be broken down into two major components, the handling of the context-free syntax, and the verification and generation of interpreter information based on the semantic rules of the parse tree automaton.

The system first processes the context-free syntax and generates a set of parse tables for use with a table driven parser. This parser is used to construct parse trees for the patterns and expressions contained in the semantic rules. These trees are then used to construct the tables which drive the interpreter (called the action tables) and are used to construct a partition of the grammar with respect to the patterns and expressions. These partitions are used to construct the underlying finite state machine for the parse tree automaton and to verify the completeness and non-redundancy of the rules.

The action tables and the underlying finite state information are then used as tables to drive the interpreter. The interpreter compares the current state, which is a parse tree of a program, against the patterns of the action tables. If the current state matches a pattern of the action table, then the corresponding expression is evaluated to yield a new state.

The system is block flowcharted in figure 5. The language design system manages six data files:

1) The Syntactic Description,
2) The Semantic Rules,
3) The Parse Tables,
4) The Action Tables,
5) The Underlying Finite State Information,
6) A Library of Test Programs,

and has five major modules which process the data:

1) The Parse Table Generator,
2) The Parser,
3) The Action Table Generator,
4) The Partition Algorithm,
5) The Interpreter.

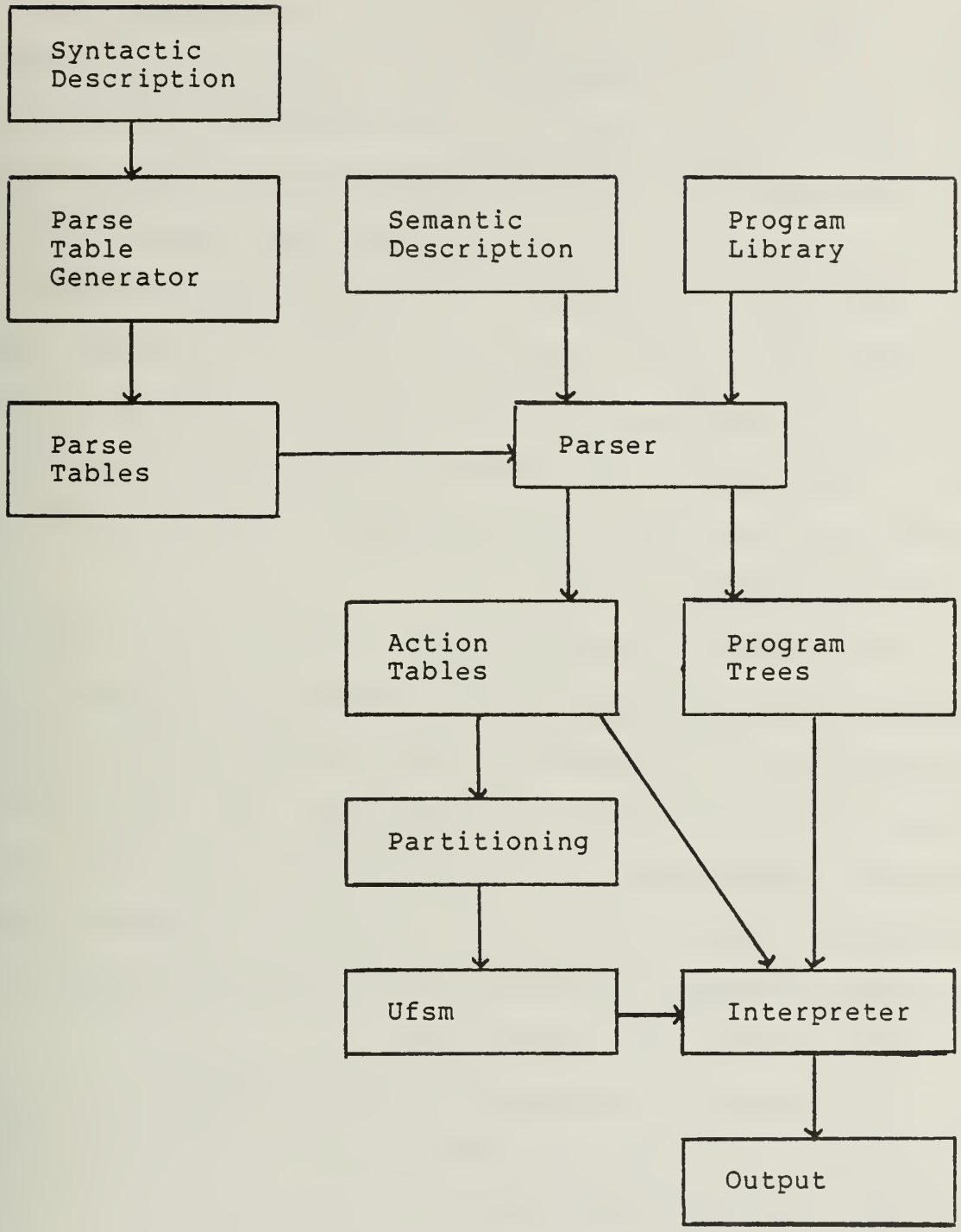
The syntactic description, the semantic rules, and the program library are maintained by a text editor. The parse tables are generated from the syntactic description using the parse table generator. The action tables are generated from the semantic rules by first parsing all the sentential forms and then by linking the patterns and the expressions together. The partition algorithm generates the underlying finite state information from the context-free syntax using the action tables as a guide. To interpret a program, a

parse tree of the program must first be constructed. Any invalid syntax is identified at this point. The parse tree is then used as input to the interpreter, the context-sensitive requirements are checked and the semantic meaning of the program is generated by interpreting the program. The results of the program may be printed, or a trace of every state the program enters may be requested.

6.2 Parsing

One major function of the language design system is the parsing of programs based on the context-free syntax. To be able to simulate a parse tree automaton, we must be able to parse sentential forms as well as programs in the language being designed. Although any type of parsing technique can be used, this implementation uses a table driven shift-reduce parser. The tables for the parser are generated using an existing parse table generator.

Since we must be able to parse sentential forms of L(G), as well as programs written in this language, we must either modify the parsing technique or we must modify the grammar. It is possible to modify the parsing technique to allow nonterminals in the input stream. This modification involves extending the parsing tables to include nonterminal symbols



Language Design System

Figure 5

where there were only terminal symbols before. For a LR(k) shift-reduce parser, we must extend the parsing table to include nonterminal symbols in the lookahead. In particular, the parsing action table must be extended to include entries for each nonterminal (Since the goto table is already defined on the union of the terminal and nonterminal symbols, it does not need to be extended to include elements from the nonterminals). This modification of the parsing tables allows the parsing of sentential forms. To parse a partition or an expression, we replace all variables names by their syntactac class name (nonterminal symbol) and then parse the resulting sentential form.

The alternative approach is to modify the grammar to include terminal symbols which represent the variables. The advantage to this technique is that we can use existing parse table generators without modifications. The disadvantage is that the added symbols and the added rules make the parsing tables slightly larger. To modify the grammar, we must introduce a rule which associates each variable name with the syntactic class. For example, if

```
val:      Operand => e | Operand Digit
```

is a rule in the syntactic description, we modify the grammar by adding an additional rule:

```
Operand => val
```

This type of rule allows us to parse the patterns and

expressions from the semantic rules. The patterns and expressions are sentential forms with the nonterminals represented by variable names. Since we have introduced new terminal symbols for each of the variable names, we may now parse the sentential forms.

Since the expressions may also contain functions, we must modify the grammar to include nonterminal nodes for these functions. If we have a function F(x) which returns elements of the syntactac class Nont, we then modify the grammar to include the rules:

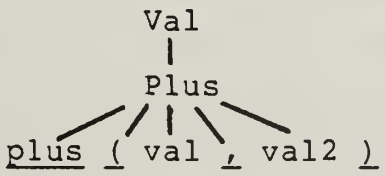
Nont -> F

F -> f (x)

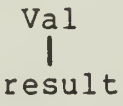
These rules introduce a new unique syntactic class F which derives the function call and a new terminal symbol f (the function name) which is unique for each function. The symbol x represents the argument list of the function and can contain nonterminal symbols as well as terminal strings. If we consider an expression which includes a function, eg.

e , Plus(val , val2) , y

then the parse tree of the expression will have a subtree of the form:



To evaluate the expression we must evaluate the function Plus and replace the subtree Plus by the result of the function call. The resulting tree for the expression will then include the subtree:



These additions to the grammar allow us to use a table driven parser. The parser must be slightly modified since we now have three types of terminal symbols:

1) xyz - strings in the programming language
2) var - variables
3) f - function names

The scanner of the parser should be able to differentiate between these different types of nodes. The parser must be able to mark the resulting nodes of the parse tree with the corresponding type. The possible node types are:

1) Nonterminal - a syntactic class name
2) Terminal - a terminal string such as xyz
3) Variable - a variable node for some syntactic class
4) Copy - a variable node which is not the first occurence of that variable. The action table generator modifies

Variable nodes into Copy nodes during the linking phase.

5) Function - an introduced nonterminal which generates a function call

The tree nodes produced by the parser are generated in preorder and have 6 fields:

Number	Semindex	Attr	Semclass	Index	Name

These fields supply information about the nodes of the parse trees. This information is sufficient to reconstruct the original sentential form, and the derivation sequence that was used to parse the it. The Number field simply identifies the node. The Semindex field gives the symbol number of the node. The Semclass field indicates the node type (nonterminal, terminal, function, variable, or copy). The Attr field is used for several different applications, depending on the type of the node. If the node is a nonterminal or a function, the Attr field gives the number of sons of the node. If the node is a variable node, the Attr field holds the value of the suffix (0 if no suffix is given). For example, the Attr field of a node corresponding to the variable val2 would have the value 2. If the node is a terminal symbol, the Attr field is not used by the interpreter but can be used by the parser to pack information about terminal symbols. For example, the value of a number

might be stored in the Attr field. The Index field is also used for several purposes. If the node is a nonterminal node, the Index field holds the number of the rule which is used to derive the sons of the node. If the node is a copy node, then the index field points back to the first occurence of that variable. The Name field supplies the label of the node of the parse tree. The Name field is included only for readability since this information can be reconstructed from the grammar and the Semindex information.

The parser is used to construct parse trees for each partition and each expression of the semantic rules. These trees are then processed by the action table generator. Additionally, the parser is used to construct parse trees for the test programs in the program library.

6.3 Action Table Generator

The action table generator prepares the action tables for the interpreter. The patterns and expressions for the transition rules given in the semantic description are first parsed. The resulting parse trees must then be 'linked' together to form tables that will drive the interpreter.

The action table generator uses the parse trees of the patterns and expressions given by the semantic rules as input. The output of the action table generator is a table of preorder traversels of the parse trees. Preorder was chosen to allow easy comparisons of the parse tree of the current state against the patterns. Additionally, by using a preorder traversal, recursive algorithms for comparing and rebuilding parse trees may be used.

The well-formedness of the transition rules are verified in three ways. First, the patterns and the expressions are parsed. This checks that both the patterns and expressions are valid sentential forms. The action table generator examines each pattern and verifies that it does not contain any function calls. Additionally, every variable which occurs in an expression must also occur in the pattern of the same rule. This requirement is checked by 'linking' the patterns and the expressions together.

In order to evaluate expressions, we must find the value of each of the variables which occur in that expression. Additionally, if the same variable occurs more than once in a pattern, then subsequent occurences of the variable will only match an identical value as the first occurence. The action table generator identifies multiple occurences of a variable and links them together. If a variable appears more than once, all occurences of that variable, except the first one, must be modified. The subsequent occurences of the variable have their semantic class changed from Variable to Copy, and a pointer to the first occurence of the variable is created in the index field. If we have a variable in an expression which does not correspond to a variable in the corresponding pattern, an error is indicated since the transition rule in not well-formed.

6.4 Verification and Optimization

The verification of a formal description is accomplished in several different modules of the language design system. The syntactic description is checked by the parse table generator. The syntactic description is used to generate a set of production rules for the context-free grammar which defines the trees of the parse tree automaton. In generating these rules, the syntactic description is checked, and variable names and function names are recognised. The generated grammar is then processed by the parse table generator and errors in the description of the grammar are identified.

The format of the semantic rules is checked during the action table generation. First each rule is parsed. This verifies that each pattern and each expression is a valid sentential form. The expression and patterns are then linked together. During this phase, we verify the requirement that all variables used in an expression must also appear in a pattern.

The final check of the semantic rules is to identify redundant rules and to look for missing rules. This is done in the partitioning phase. A partition is constructed with respect to the patterns and expressions of the semantic rules. The underlying finite state information is then

generated. In generating this information, any redundant rules are identified. Additionally, any partitions which do not correspond to a pattern are found. These unmatched partitions indicate that the semantic rules are incomplete.

The partitioning phase also produces the underlying finite state information. This information is used in the interpreter to eliminate all unnecessary comparisons. This optimises the interpreter by removing all unnecessary matching.

6.5 Interpreter

The interpreter interprets programs in the language under design by simulating a parse tree automaton. The initial state of the interpreter is the parse tree of a program from the program library together with its internal data. This tree is compared against the patterns of the transition rules and the next state is constructed by evaluating the expression of the first rule to match.

6.5.1 Matching

The current state, which is a parse tree of a program in the language under design, is matched against all possible transition rules. A match is successful if the pattern is a section of the parse tree of the current state. The match is done in a recursive manner starting with the root of the current state and the first node of the preorder list of the pattern. There are three cases based on the type of the pattern node. These cases are:

Terminal - Match only if the tree node is the an identical terminal.

Nonterminal - Match if the current tree node is the same nonterminal and if all the sons of the tree node match the sons of the pattern (this is a recursive call).

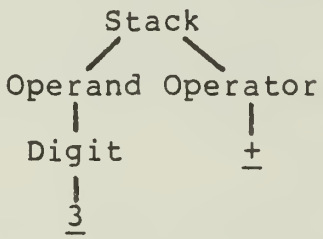
Copy - Match only if the current tree is identical with the tree which first matched this variable. The pointer field of the copy node points to the first occurence of the variable which in turn points to the tree which first matched.

Matching terminal nodes is straightforward. Matching copy nodes is also easy, as we must simply check that the current subtree is identical whith the value which matched the first

occurence of the variable. When we match the sons of a nonterminal, the son can be a variable. Since the variable represents any tree of the corresponding syntactic class, the match succeeds and the value of the current subtree is saved. For example, if a subpart of the pattern is:

```
Stack stk
```

and the corresponding subtree of the current state is:



then the match is successful and the value of the subtree is bound to the variable var (a pointer to the subtree is saved). Subsequent occurences of the variable must have the copy attribute and the value of the subtree will be matched against the saved value of the variable.

6.5.2 Next State Construction

Once the matching rule is found, a new state is calculated by evaluating the expression. A new parse tree is constructed using the expression as a template. Any expression nodes whose attributes are 'copy' are replaced by the value of the corresponding variable. Any functions are evaluated possibly by calling the interpreter to evaluate a submodule. For example, if the preorder list for the expression is:

Val, Plus, plus, (, val, ,, val2,)

then when we evaluate the expression, we first replace the variables val1 and val2 by their values (1 and 2) and then evaluate the function Plus(val,val2). The resulting tree is used as the value of the nonterminal node, Val. The resulting subtree would be

since the result of evaluating plus(1,2) is the the string 3.

6.6 Example

Let us consider the example of a pocket calculator. The formal description of such a calculator consists of four parts, the syntactic description, the semantic module for the calculator, the semantic module which defines the function Times, and the built in function Plus. This example was chosen to show both types of function calls, the defined module, and the built in function.

The semantic description for these modules is:

```
        State    => Calcstate | Plusstate
          Calcstate => Stack ',' Display ',' Input
          Plusstate => Operand ',' Operand ',' Operand
                e | Operand
dis:      Display => Operand
stk:      Stack => e | Operand Operator
op:          Operator => '+' | '*'
val:         Operand => e | Operand Digit
y:        Input => e | Key Input
key:         Key => Digit | Operator
digit:          Digit => '0' | '1' | '2' | '3' | '4'
                digit => '5' | '6' | '7' | '8' | '9'
          Operand<=plus( Operand ',' Operand )
          Operand<=times( Operand ',' Operand )
```

The last two rules in the semantic description define the functions Plus and Times and bind them to the semantic class Operand. This description will generate the following grammar. Note that new rules have been introduced for every variable and every function.

```
Calcstate =>   Stack , Display , Input
Plusstate =>   Operand | Operand , Operand , Operand
Display   =>   Operand | dis
Stack     =>   e | Operand Operator | stk
Operator  =>   + | * | op
Operand   =>   e | Operand Digit | Plus | Times | val
Plus      =>   plus ( Operand , Operand )
Times     =>   times ( Operand , Operand )
Input     =>   e | Key Input | y
Key       =>   Digit | Operator | key
Digit     =>   0 | 1 | 2 | 3 | 4
             | 5 | 6 | 7 | 8 | 9
```

For example, the rule 'Display => dis' was introduced since dis is a variable bound to the syntactic class Display. Additionally, the rules

```
Operand => Plus
Plus    => plus ( Operand , Operand )
```

are introduced to define the function plus which returns elements of the syntactic class Operand. This grammar is then used by the parse table generator to generate the parse tables. These tables will be used by the parser to parse programs in the language and to parse the semantic rules.

For this example, the semantic rules consist of two modules, Calc and times. Both of these modules use the built in function Plus. The module Times also uses the function Sub. The transition rules are written on two lines, with the name of the rule and the sentential form of the pattern written on the first line, and the sentential form of the expression written on the second. The semantic modules for Calc and Plus are listed in figure 6.

```
module calc

calc1:    stk , val , digit y                    ->
          stk , val digit , y

calc2:    , val , op y                           ->
          val op , , y

calc3:    val + , val2 , y                       ->
          , plus ( val , val2 ) , y2 ) , y

calc4:    val * , val2 , y                       ->
          , times ( val , val2 , 0 ) , y

return:   , val ,                                ->
          , val ,

error:    val op , , op2 y                       ->
          , val , y

start:    , , y                                  ->
          , , y

endmod


module times

start:    val , val2 , 0                         ->
          val , val2 , 0

return:   val , 0 , val2                         ->
          val2

times:    val , val2 , val3                      ->
          val , sub( val2 , 1 ) , plus( val , val3 )

endmod
```

Semantic Modules

Figure 6

There are three special rule names used in both modules. The name 'start' indicates that the module will have an initial state which matches the pattern of this rule. This rule is not included in the action tables but is only used in generating the underlying finite state information. The first match will only match rules which match the rule 'start'. Another rule which has a special name is 'return'. This rule is included in the action tables, but after applying this rule, no other rules are tried. Therefore, this rule forces a halt state in the underlying finite state machine and causes the machine to return to the calling module. The value returned is the parse tree which is the evaluation of the expression of the rule 'return'. A rule with a name 'error' also causes a halt state in the underlying machine. However, an 'error' rule causes the interpreter to stop executing. The error rules are used to report programs that do not satisfy the context-sensitive requirements.

The rules in each module are parsed and linked together by the action table generator. This forms a set of tables which are used by the interpreter. For example, the table entry for the rule calcl is shown in figure 7.

calcl:

1	25	5	nont	1	Calcstate
2	26	1	nont	8	Stack
3	3	Ø	var	Ø	stk
4	1	Ø	ter	Ø	'
5	27	1	nont	4	Display
6	3Ø	1	nont	16	Operand
7	7	Ø	var	Ø	val
8	1	Ø	ter	Ø	'
9	28	2	nont	2Ø	Input
1Ø	35	1	nont	22	Key
11	32	1	nont	35	Digit
12	1Ø	Ø	var	Ø	digit
13	28	1	nont	21	Input
14	8	Ø	var	Ø	y

expression:

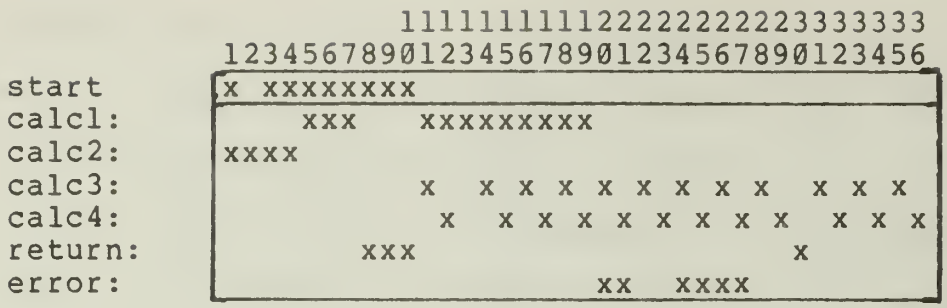
1	25	5	nont	1	Calcstate
2	26	1	nont	8	Stack
3	3	Ø	copy	3	stk
4	1	Ø	ter	Ø	'
5	27	1	nont	4	Display
6	3Ø	2	nont	13	Operand
7	3Ø	1	nont	16	Operand
8	7	Ø	copy	7	val
9	32	1	nont	35	Digit
1Ø	1Ø	Ø	copy	12	digit
11	1	Ø	ter	Ø	'
12	28	1	nont	21	Input
13	8	Ø	copy	14	y

Action Table Entry for Rule Calcl

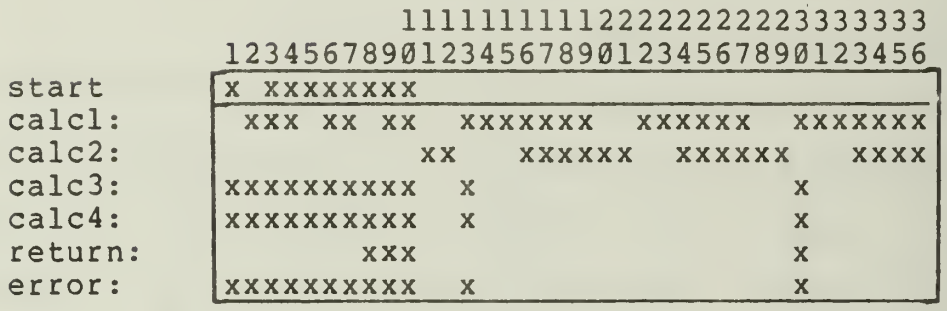
Figure 7

The semantic rules and the syntactic descriptions are also used as input to the partition generator. The grammar is refined with respect to the partitions and expressions of each module. For example the module Calc produces 36 partitions. Each partition and each expression of the module is the union of a subset of the partition. The composition of each pattern and expression of the module Calc is shown in figure 8, while the underlying finite state machine of the module is shown in tabular form in figure 9.

The partition elements of each partition and expression are graphicaly represented in the configuration matricies shown in figure 8. For example, we can see that the pattern of the first transition rule, calcl, is composed of partitions 5 through 7 and 11 through 19. This rule correspond to state<1> of the underlying finite state machine. Since the starting configuration includes partition 5, the rule calcl will be one of the rules matched against the first state. Therefore, state<1> will be a successor state of the initial state, state<Ø>, of the underlying finite state machine. Since the expression of rule calcl includes partitions which are in the patterns of every other rule, it is possible to apply any rule after applying rule calcl.



Partition Configuration



Expression Configuration

Partition Composition of the Patterns
and Expressions

Figure 8

If we consider the underlying finite state machine of the module Calc, we can see that the rules initially attempted are rules 1, 2, and 5 which correspond to the rules calc1, calc2, and return. These three rules are the rules whose patterns contain partitions that are also contained by the expression of the initial configuration described by the rule start. Note that the rule start produces the initial state, state<Ø>, of the underlying finite state machine. The states corresponding to the rules return and error do not have any successors. Instead, a return is indicated by a -1 and an error is indicated by a -2. If after evaluating an expression, the only successor state is -1, then the current value is returned as the result of a function call. If the successor state is -2, then an invalid state has resulted and an error is signaled. The underlying finite state information is used by the interpreter to choose which rules to attempt to match against a current state. If we have a current state which corresponds to state<i> of the underlying machine, then we only need test those rules which correspond to the successor states of state<i>. For instance, if we had just applied the rule calc4, the corresponding state of the underlying machine would be state<4>. Therefore, the only rules we would need to consider would be rules calc1, calc2, and return. Indeed, if we have applied rule calc4, then the current state is derived from the sentential form ' ,times(val , val), y' and hence will match only the patterns

start:	0:	1	2			5	
calc1:	1:	1	2	3	4	5	6
calc2:	2:	1		3	4		6
calc3:	3:	1	2			5	
calc4:	4:	1	2			5	
return:	5:	-1					
error:	6:	-2					

Underlying Finite State Machine

for the Module Calc

Figure 9

stk , var , digit y

, val , op y

, val ,

depending on the value of the remaining input. If the remaining input is empty, the rule return will be matched. If the remaining input starts with a digit, the rule calcl will be used to shift the digits onto the display. In the only remaining case, the remaining input starts with an operater and the display and the operator will be pushed onto the stack.

Finally, let us consider the operation of the interpreter. Figure 1Ø shows the computation sequence of the calculator with the initial configuration of ',,2+3+4'. Only the leaves of the parse trees are shown. Actually, each state is the parse tree of the terminal strings shown in figure basfig+4. Calls to the built in functions are simply evaluated. However, the call to the function Times is evaluated using a recursive call to the interpreter. When the function Temes(5,4,Ø) is called, the interpreter is initialised to the state '5,4,Ø'. A calculation sequence is then calculated using the rules from module Times. When the state '5,Ø,2Ø' is reached, the return rule is matched, and the value '2Ø' (which is an Operand) is returned. This value is then used as the value of the calculators display in the expression of rule calc4. When the input of the calculator

is exhausted, the state '_, val _,' is matched by the rule return. Since this is the top level module, the execution will halt with the final state '_,20_,'.

Stack	Display	Input
	,	,2+3*4
	,2	+3*4
2+	,	,3*4
2+	,3	*4
	,5	,*4
5*	,	*4
5*	4	,

call the module times(5,4,0)

5	,4	,0
5	,3	,5
5	,2	,10
5	,1	,15
5	,0	,20
	20	

return from module times

,20	,

Interpreter Evaluation of ,,2+3*4

Figure 10

CHAPTER 7

CONCLUSIONS

The language design system that is described in the introduction and described in chapter 6 has been implemented. This system has been used to design and implement 'toy' languages of the complexity of the pocket calculator. The design system allows the user to generate a working interpreter for simple languages with only a few hours work. For example, the description of the pocket calculator takes about one hour to develop. The interpreters generated in this way do not seem to suffer from limitations in size or speed. However, interpreting larger languages such as PL/1, will probably be too inefficient for continued use. Once the formal specification of a large language is developed and verified, a compiler can then be designed following the formal specification.

In the case of larger languages, the language design system helps the designer specify the formal specification of the language. This system has been used to verify the formal specification of a block structured language, SYBIL. This specification contains over 150 syntactic rules, and over 100 semantic transition rules. This description was of such complexity that the mechanical verification caught several errors which escaped human detection. Once these errors were detected, it was a simple matter to change the specification to correct these errors. The language design system has also been used to verify parts of a formal definition of the programming language Asple.

There are several different possible extensions of the langage design system. It may be possible to extend the interpreter into a syntax directed parser by adding a register for the object code. Each transition rule would then generate a sequence of machine instructions and append them to the existing code. For example, we might generate the addition operation in the following manner:

(num1 $\pm$ num2 x , pgm) ->

(x , pgm Push(num1) Push(num2) Add)

Once we recognise an addition, we append the code to push the operands onto the stack and then add them together. This type of code generation would be more powerful than a strict syntax directed translation since we can call subfunctions

and can manipulate the source program and the object code using the parse tree automaton.

Other extensions are possible in the language design system. In the current system, all parsing is done with a table driven parser. The current parser will only parse sentential strings which are derived from the start symbol of the grammar. Since each function can be described in its own module, we would like to be able to generate parse trees whose root node corresponds to an arbitrary syntactic class. A table driven parser with this capability could be generated by modifying the parse tables to accept arbitrary starting symbols. Alternatively the grammar may be augmented to include a new start symbol which derives every nonterminal in the grammar.

The language design system may also be extended by offering the language designer more aids to help in the design process. Some possible aids are libraries of common functions, either machine coded routines for such operations as addition, or predefined semantic modules to do common operations such as remove blanks. Also, string matching primitives could be added to the patterns and expressions to aid in writing the rules. For example, a matching function, *rem*, would be useful in matching the end of a pattern. If we had a sentential form like

x y z a b c d e

and are only interested in matching x y z, we might write

x y z *rem*.

Here the function *rem* would automatically generate all possible remainders of the string x y z. Other possible string matching functions include *arb* and *len(n)* for matching arbitrary strings or strings of a fixed length. Indeed, the parse tree automaton provides an efficient method of performing a series of predifined string matches. When it is used in this mannar, it out performs the string matching language Snobol. Of course, the class of problems which the parse tree automaton solves is only a subset of those problems which may be solved using Snobol.

The partitioning algorithm and the parsing algorithm may be merged into one routine. We could use the partitions to generate a top down parse of the sentential forms. We could also extend the parser/partitioner by adding editing functions to allow easy changes to the semantic rules.

LIST OF REFERENCES

Backus J. W. [1959] "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," Proceedings of the International Conference of Information Processing, UNESCO, pp. 125-132.

Chomsky, N. [1956] "Three Modles for the Description of Languages," PGIT, 2:3, pp. 113-124.

Department of Defense [1960] COBOL: Initial Specifications for a Common Business Oriented Language, U. S. Govt. Printing Office, Washington, D.C.

Feyock, S. [1975] "Toward an Implementation of the Vienna Definition Language," Proceedings 1975 International Conference on ALGOL68, pp. 370-384.

Garwich, J. V. [1966] "The Definition of Programming Languages by Their Compilers," Formal Language Description Languages for Computer Programming (proc. IFIP Working Conf. 1964) (Steel, T. B., Ed.). North-Holland Publ. Co. (Amsterdam) pp. 266-294.

Greibach, S. A. [1965] "A new Normal Form Theorm for Context-Free Phrase Structure Grammars," JACM 12:1, pp. 42-52.

Hoare, C. A. R. [1969] "An Automatic Basis for Computer Programming," Comm. ACM 12:10, pp. 576-580.

Hoare, C. A. R. [1974] "Consistent and Complementary Formal Theories of the Semantics of Programming Languages," Acta Informatica 3, pp. 135-153.

Irons, E. T. [1970] "Experience with an Extensible Language" Comm. ACM 13:1, pp. 31-40.

Kampen, G. R. [1973] SIBYL: A Formally Defined Interactive Programming System Containing an Extensible Block-Structured Language. (Ph.D. Thesis) Tech. Rept. #73-06-16, Computer Science Group, University of Washington (Seattle).

Kampen, G. R. and J. L. Baer [1975] "The Formal Definition of Semantics by String Automata" Computer Languages V 1. pp. 121-138.

Ledgard, H. F. [1977] "Production Systems: a Notation for Defining Syntax and Translation," IEEE Transactions on Software Engineering, Vol SE-3, No.2, pp. 105-124.

Leuis P. M. and Stearns [1968] ,"Syntax-Directed Translation," JACM 15, pp. 3-9.

Lucas P., P. Lauer, and H. Stigleituer [1968] "Method and Notation for the Formal Definition of Programming Languages," IBM Technical Report 25.078 IBM Lab., Vienna, Austria.

Lucas P. and K. Walk [1969] "On the Formal Description of PL/1," Annual Review of Automatic Programming 6:3 pp. 105-182.

Marcotty M., H. Ledgard, and G. Bachmann [1976] "A Sampler of Formal Definitions," Computing Surveys 8:2 pp. 155-276.

Tennent R. D. [1976] "The Denotational Semantics of Programming Languages," CACM 19:8 pp. 437-453.

van Wijngaarden, A., Mailloux, B. J., Peck, J. E., and Koster, C. H. A. [1969] Report on the Algorithmic Langauge: ALGOL68 MR 101, Mathematisch Centrum, Amsterdam, The Netherlands.

Wegner P. [1972] "The Vienna Deffinition Langague," Computing Surveys 4:1 pp. 5-63.

VITA

Brian Alfred Hansche was born on October 3, 1950 in Albuquerque New Mexico. He attended Highland High School in Albuquerque from which he graduated in June, 1969. He then attended the University of New Mexico and graduated magna cum laudi with a B. S. in Mathematics in 1971. Mr. Hansche then begin his graduate work at the University of Illinois where he recieved his Masters degree in 1976. During the time spent working on his Masters Degree; and while working on his Ph. D. degree, he was employed as a Reasearch Assistant for the Department of Computer Science. Mr. Hansche has also served as a Teaching Assistant for the Department of Computer Science at the University of Illinois.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-78-935	2.	3. Recipient's Accession No.

4. Title and Subtitle

AN IMPLEMENTATION OF A SYSTEM FOR THE FORMAL DEFINITION OF PROGRAMMING LANGUAGES

5. Report Date: August 1978

6.

7. Author(s)

Brian Alfred Hansche

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts This paper describes a method for generating a table-driven interpreter for a programming language from a formal specification of its syntax and semantics. Such interpreters would be useful in verifying the correctness of formal specifications, and in providing experience with initial versions of experimental languages. The paper discusses existing formal specification methods and selects one method, based on a string replacement mechanism, as the basis for implementing a table-driven interpreter. A class of machines called Parse Tree Automata is defined. These machines are such that each state can be represented as a parse tree of a concrete program. An interpreter is then defined by a computation sequence of the Parse Tree Automaton. A method of constructing a table-driven interpreter based on these abstract machines is given and algorithms for reducing the number of transitions needed by the interpreter are supplied. The paper also includes a method of verifying that the formal specification is complete, well formed, and not redundant.

17. Key Words and Document Analysis. 17a. Descriptors

formal languages, programming languages, syntax, semantics, interpreters

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
	20. Security Class (This Page) UNCLASSIFIED	22. Price